ABSTRACT

GHOLKAR, NEHA. On the Management of Power Constraints for High Performance Systems. (Under the direction of Frank Mueller).

The supercomputing community is targeting exascale computing by 2023. A capable exascale system is defined as a system that can deliver 50X the performance of today's 20 PF systems while operating in a power envelope of 20-30 MW [Cap]. Today's fastest supercomputer, Summit, already consumes 8.8 MW to deliver 122 PF [Top]. If we scaled today's technology to build an exascale system, it would consume 72 MW of power exceeding the exascale power budget. Hence, intelligent power management is a must for delivering a capable exascale system.

The research conducted in this dissertation presents power management approaches that maximize the power efficiency of the system. Power efficiency is defined as performance per Watt. The proposed solutions achieve improvements in power efficiency by increasing job performance and system performance under a fixed power budget. We also propose a fine-grained, resource utilization-aware power conservation approach that opportunistically reduces the power footprint of a job with minimal impact on performance.

We present a comprehensive study of the effects of manufacturing variability on the power efficiency of processors. Our experimentation on a homogeneous cluster shows that there is a visible variation in power draw of processors when they achieve uniform peak performance. Under uniform power constraints, this variation in power translates to a variation in performance rendering the cluster non-homogeneous even under uniform power bounds. We propose Power Partitioner (PPartition) and Power Tuner (PTune), two variation-aware power scheduling approaches that in co-ordination enforce system's power budget and perform power scheduling across jobs and nodes of a system to increase job performance and system performance on a power-constrained system. We also propose a power-aware cost model to aid in the procurement of a more performant capability system compared to a conventional worst-case power provisioned system.

Most applications executing on a supercomputer are complex scientific simulations with dynamically changing workload characteristics. A sophisticated runtime system is a must to achieve optimal performance for such workloads. Toward this end, we propose Power Shifter (PShifter), a dynamic, feedback-based runtime system that improves job performance under a power constraint by reducing performance imbalance in a parallel job resulting from manufacturing variations or non-uniform workload distribution. We also present Uncore Power Scavenger (UPS), a runtime system that conserves power by dynamically modulating the uncore frequency during the phases of lower uncore utilization. It detects phase changes and automatically sets the best uncore frequency for every phase to save power without significant impact on performance.

On the Management of Power Constraints for High Performance Systems

by
Neha Gholkar

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2018

APPROVED BY:

Vincent Freeh	Harry Perros
Huiyang Zhou	Frank Mueller Chair of Advisory Committee

DEDICATION

This dissertation is dedicated to my parents, Bharati Gholkar and Pandharinath Gholkar, for their endless love, support and encouragement and to my uncle, Rajendra Shirvaikar, for introducing me to computers and inspiring me to become an engineer at a very young age.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Frank Mueller, for giving me the opportunity to work with him and for providing encouragement, guidance and constant support over the years. Frank taught me the process of research, critical thinking and provided constructive feedback in each of our meetings. Frank gave me the freedom and independence of pursuing new research ideas which in turn gave me the confidence of conceptualizing and proposing bold ideas and taking them to conclusion. I am grateful for his support and patience when things didn't go as planned. Finally, I would also like to thank him for imparting timeless life lessons such as, "persistence always pays" and "hope for the best and prepare for the worst".

I want to thank Barry for providing me the opportunity to work with him at the Lawrence Livermore National Laboratory. Having access to a large-scale production cluster at Livermore helped me understand the real world challenges in supercomputing. Barry taught me the art of visualization when it came to analyzing large datasets and subsequently, the process of data-driven idea generation. Barry, I cannot thank you enough for introducing me to R and the non-conventional style of plotting data. I will always remember our scientific discussions on paper napkins at restautants and how awesome I felt about the idea of being a computer scientist in those moments. Lastly, I would like to thank you for your relentless support and encouragement over the past few years.

I would like to thank my committee members, Dr. Vincent Freeh, Dr. Harry Perros, and Dr. Huiyang Zhou, for their timely feedback and suggestions on my research.

I would like to thank all my awesome labmates in the Systems Research Laboratory at NCState as well as at Lawrence Livermore National Laboratory. I would like to thank all the FRCRCE friends, especially Abhijeet Yadav and Amin Makani, for their motivation and kind words. I would like to thank Prajakta Shirgaonkar, Mandar Patil, Aparna Sawant and Ajit Daundkar for being the constant support.

I wish to thank the four pillars of my life, Bharati Gholkar, Pandharinath Gholkar, Neela Shirvaikar and Rajendra Shirvaikar, who have been my strength for as long as I have lived. Meghana and Amit, I wouldn't be here without you. Thanks for your unwavering support and motivation. Ishani and Yohan, you are little but your unconditional love has lifted me higher every time.

Last but not the least, I want to thank Jimit Doshi for inspiring and motivating me throughout the process. You have taught me how to find the silver linning in every situation. You have been my greatest critic and my strongest support at the same time. This journey, with its ups and downs, would not have been the same without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1 INTRODUCTION

The supercomputing community is headed toward the era of exascale computing, which is slated to begin around 2023. Today's fastest supercomputer, Summit, consumes 8.8 MW to deliver 122 PF [Top]. If we scaled today's technology to build exascale systems, they would consume 72 MW of power leading to an unsustainable power demand. Hence, to maintain a feasible electrical power demand, the US DOE has set a power constraint of 20-30 MW on future exascale systems. In order to deliver an exaflop under this constraint, at least an order of magnitude improvement in performance with respect to today's systems needs to be achieved while operating under the power envelope of 20-30 MW [Ber08; Dal11; Ins; Sar09; Cap]. Toward this end, the semiconductor industry is focussing on designing efficient processing units mainly by means of developments in process technology as well as in architecture. The supercomputing research needs to focus on using this hardware efficiently by developing novel system software solutions to manage power and improve performance.

The research conducted in this dissertation proposes power scheduling solutions aimed at improving performance of applications as well as system performance on a power-constrained system.

1.1 Challenges and Hypothesis

The challenges in power and performance for High Performance Computing (HPC) can be summarized as follows:

- Power that can be brought into the machine room is limited.
- After the initial burn-in phase (LINPACK execution), the procured power capacity is never utilized again [Pat15].
- Job performance is degraded by factors such as performance variation and imbalance, and suboptimal resource utilization.
- To achieve exascale, a 50X performance improvement is needed with no more than a 3.5X increase in power relative to today's 20 PF systems.

This work proposes power management approaches that make power a first-class citizen in resource management. The hypothesis of this dissertation can be stated as follows:

To design power-efficient systems, power needs to be managed discretely at both system-level and job-level to both improve performance under a power constraint and to reduce wasteful consumption of power that does not contribute to performance. Toward this end, it is crucial to identify and reduce inefficiencies in the system with respect to performance imbalance and suboptimal resource utilization.

1.2 Contributions

This work contributes a novel approach of enforcing a system-level power budget and a variation-aware job-power scheduler that improves job performance under a power constraint. We also propose a runtime system that dynamically shifts power within a parallel job to reduce imbalance and to improve performance. We investigate the impact of Uncore Frequency Scaling (UFS) on performance and propose a runtime system that dynamically modulates the uncore frequency to conserve power without a significant impact on performance.

1.2.1 Power Tuner and Power Partitioner

A power-constrained system operates under a strict operational power budget. A naïve approach of enforcing a system-level power constraint for a system is to divide the distribute power uniformly across all its nodes. However, under uniform power bounds, the system is no longer homogeneous [Rou12; Gho16]. There are many potential root causes of this variation, including, but not limited to, process variation and thermal variation due to ambient machine room temperature. Scheduling jobs on such a non-homogeneous cluster presents an interesting problem.

We propose a 2-level hierarchical variation-aware approach of managing power at the machine-level. At the macro level, *PPartition* partitions a machine's power budget across jobs to assign a power budget to each job running on the system such that the machine never exceeds its power budget. At the micro level, *PTune* makes job-centric decisions by taking the performance variation into account. For every moldable job (number of ranks is modifiable), PTune determines the optimal number of processors, the selection of processors and the distribution of the job's power budget across them, with the goal of maximizing the job's performance under its power budget. PTune achieves a job performance improvement of up to 29% over uniform power. PTune does not lead to any performance degradation, yet frees up 40% of the resources compared to uniform power. PPartition and PTune together improve the throughput of the machine by 5-35% compared to conventional scheduling. The limitation of the proposed solution is that it relies on the off-line characterization data to make decisions before the beginning of job execution.

1.2.2 Power Shifter

Most production-level parallel applications suffer from computational load imbalance across distributed processes due to non-uniform work decomposition. Other factors like manufacturing variation and thermal variation in the machine room may amplify this imbalance. As a result, some processes of a job reach blocking calls, collectives or barriers earlier and then wait for others to reach the same point in execution. Such waiting results in a wastage of energy and CPU cycles that degrades application efficiency and performance.

We propose Power Shifter (PShifter), a runtime system that maximizes a job's performance without exceeding its assigned power budget. Determining a job's power budget is beyond the scope of PShifter. PShifter takes the job power budget as an input. PShifter is a hierarchical closed-loop feedback controller that makes measurement-based power decisions at runtime and adaptively. It does not rely on any prior information about the application. For a job executing on multiple sockets (where a socket is a multicore processor or a package), each processor is periodically monitored and tuned by its local agent. A local agent is a proportional-integral (PI) feedback controller that strives to reduce the energy wastage by its socket. For an "early bird" that waits at blocking calls and collectives, it lowers the power of the socket to subsequently reduce the wait time. The cluster agent oversees the power consumption of the entire job. The cluster agent senses the power dissipation within a job in its monitoring cycle and effectively redirects the dissipated power to the sockets on the critical path to improve the overall performance of the job (i.e., shorten the critical path).

Our evaluations show that PShifter achieves a performance improvement of up to 21% and energy savings of up to 23% compared to the naïve approach. Unlike prior work that was agnostic of phase changes in computation, PShifter is first to transparently and automatically apply power capping non-uniformly across nodes of a job in a dynamic manner adapting to phase changes. It could readily be deployed on any HPC system with power capping capability without any modifications to the application's source code.

1.2.3 Uncore Power Scavenger

Chip manufactures have provided various knobs such as Dynamic Frequency and Voltage Scaling (DVFS), Intel's Running Average Power Limit (RAPL) [Int11], and software controlled clock modulation [Int11] that can be used by system software to improve power efficiency of systems. Various solutions [Lim06; Rou07; Rou09; Fre05b; Bai15; HF05; Ge07; Bha17] have been proposed that use these knobs to achieve power conservation without impacting performance. While these solutions focussed on the power efficiency of cores, they were oblivious of the uncore, which is expected to be a growing component in future generations of processors [Loh].

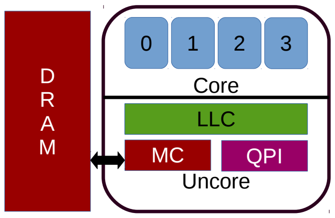


Figure 1.1 A processor is a chip consisting of core and uncore. The uncore consists of memory controllers (MC), Quick Path Interconnect (QPI) and the last level cache (LLC).

Fig. 1.1 depicts the architecture of a typical Intel server processor. A chip or a package consists of two main components, core and uncore. A core typically consists of the computation units (e.g., ALU, FPU) and the upper levels of caches (L1 and L2) while the uncore contains the last level cache (LLC), the Quick Path Interconnect (QPI) controllers and the integrated memory controllers. With increasing core count and size of LLC and more intelligent integrated memory controllers on newer generations of processors, the uncore occupies as much as 30% of the die area [Hil], significantly contributing to the processor's power consumption [Gup12; SF13; Che15]. The uncore's power consumption is a function of its utilization, which varies not only across applications but it also varies dynamically within a single application with multiple phases. We observed that Intel's firmware sets the uncore frequency to its maximum on detecting even the slightest uncore activity resulting in high power consumption. Replacing such a naïve scheme with a more intelligent uncore frequency modulation algorithm can save power. Toward this end, we propose Uncore Power Scavenger (UPS), a runtime system that automatically modulates the uncore frequency to conserve power without significant performance degradation. For applications with multiple phases, it automatically identifies distinct phases spanning from CPU-intensive to memory-intensive and dynamically resets the uncore frequency for each phase. To the best of our knowledge, UPS is the first runtime system that focusses on the uncore to improve power efficiency of the system.

Our experimental evaluations on a 16-node cluster show that UPS achieves up to 10% energy savings with under 1% slowdown. It achieves 14% energy savings with a worst-case slowdown of 5.5%. We also show that UPS achieves up to 20% speedup and proportional energy savings compared to Intel's RAPL with equivalent power usage making it a viable solution for power-constrained computing.

CHAPTER

2 BACKGROUND

This chapter is structured as follows: Section 2.1 presents background information about a typical HPC center and the HPC workloads. Section 2.2 describes “hardware-overprovisioning“, one of the key foundational ideas of this research. Section 2.3 provides information about power measurement and control mechanisms on state of the art server systems.

2.1 Fundamentals of High Performance Computing

An HPC system is a powerful parallel computing system that is used to solve some of the most complex problems. It is used in various domains, including, but not limited to finance, biology, chemistry, data science, physics, computer imaging and recognition. HPC users are typically scientists and engineers utilizing HPC resources for applications such as defense and aerospace work, weather and climate monitoring and prediction, protein folding simulations, urban planning, oil and gas discovery, big data analytics, financial forecasting, etc.

2.1.1 Architecture

An HPC system is a collection of several server nodes connected by a high-bandwidth low-latency network interconnect. It mainly consists of two types of nodes, login nodes and compute nodes. Users access HPC resources via login nodes. Login nodes are shared by multiple users. A user can start an interactive session on a login node for development purposes or for pre-processing or post-processing tasks. Users submit one or more batch jobs requests (consisting of the scientific workload) to the job queue from the login node.

HPC jobs execute on one or more compute nodes. Compute nodes are connected to a shared data storage. Each compute node mainly consists of one or more sockets hosting high-end multicore processors, such as Intel Xeons, the memory subsystem, and network interface cards (NIC) connecting the node to a high-bandwidth low-latency network such as Inifiniband. It may also have additional accelerators such as Graphics Processing Units (GPUs). Figure 2.1 depicts the architecture of an HPC system.

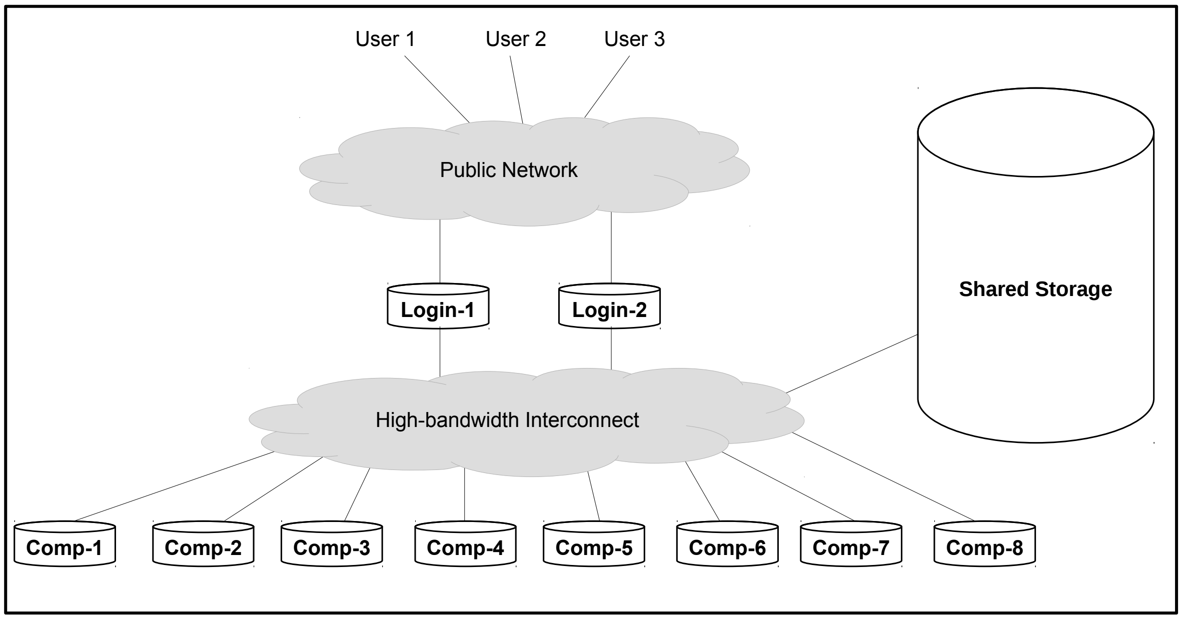
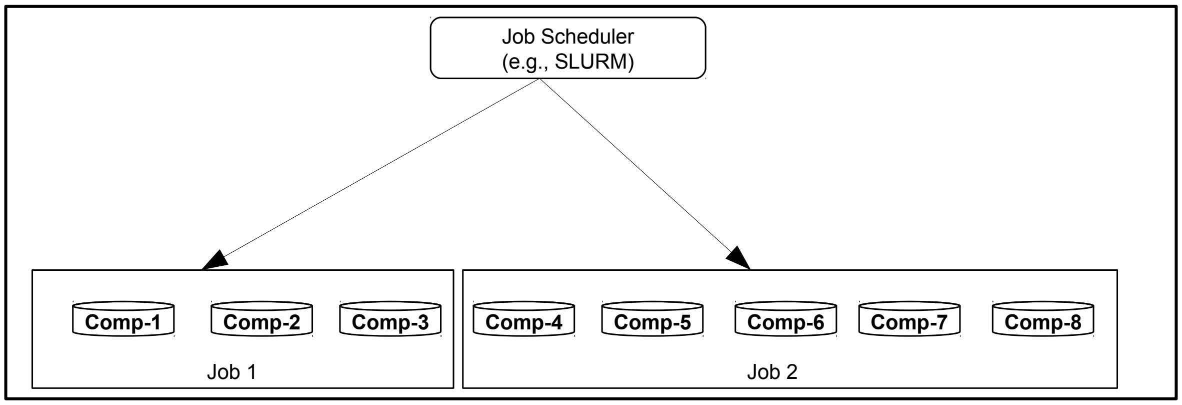


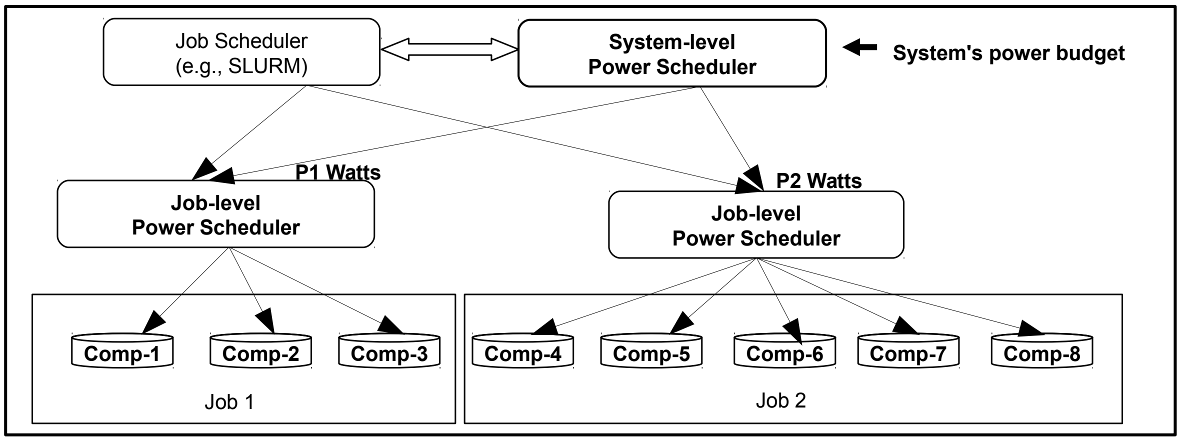
Figure 2.1 HPC architecture

2.1.2 Coordinated job and power scheduling

When a job request is submitted by a user, it is enqueued to a job queue. A job request mainly consists of information such as the link to the binary executable, application inputs and the resource request, which includes the required number of compute nodes and the expected duration for which the nodes need to be reserved during job execution. A job scheduler such as SLURM [Yoo03] then determines which of the waiting jobs to dispatch depending on several factors such as job priorities, resource request and availability. Each dispatched job is allocated a dedicated set of nodes for the requested duration. In other words, compute nodes are not shared between jobs. This is



(a) Conventional job scheduling with dedicated compute nodes (Comp-X)



(b) Coordinated job and power scheduling

Figure 2.2 Resource management on future exascale systems

depicted in Figure 2.2 (a).

As future HPC systems are expected to be power-constrained, power management will be one of the critical factors to delivering a capable exascale system. Hence, to manage power discretely a hierarchical power management framework will be employed alongside the conventional job scheduler on future systems. At the top-level, a system-level power scheduler needs to monitor the power consumption of the whole cluster and ensure that the aggregate power consumption of the system never exceeds the system's power budget. It can achieve this objective by assigning job power budgets to each of the jobs executing on the system in parallel such that the total power consumption of all the jobs never exceeds the system's power budget. A job-level power scheduler per job further monitors the power consumption of all the resources allocated to that job. It is within the purview of the job-level power scheduler to ensure that the total power consumption of the job never exceeds its power budget (e.g., the power consumption of the job on comp1-comp3 never exceeds P1 Watts). The job-level power scheduler may then distribute the job's power budget across all the resources (e.g., nodes) of the job. The total power consumption of a job is the aggregation of power consumed by the dedicated (i.e., nodes) and the shared (e.g., network, routers) hardware components on which it executes, of which the nodes are the largest contributors. Power consumed by other facility level resources such as water cooling cannot be managed at job-level granularity. Hence, we focus on power consumed by nodes in this work.

2.1.3 A Uniform Power Scheme

A naïve approach of enforcing a system's power budget is to distribute the budget *uniformly* across all the nodes of the system. We call this uniform power (UP). UP can be enforced by statically constraining the power consumption of nodes to $\frac{System'sPowerBudget}{N}$, where N is the number of nodes in the system. A similar approach can be employed to enforce a job power budget. At job-level, UP can be enforced by statically constraining the power consumption of nodes to $\frac{Job'sPowerBudget}{N_{job}}$, where N_{job} is the number of nodes of the job. While UP enforces system-level or job-level power bounds it leads to sub-optimal performance. The disadvantages of UP will be discussed in more detail in Chapters 3-5.

2.2 Hardware-overprovisioning

Exascale systems are expected to be power-constrained: the size of the system will be limited by the amount of provisioned power. Existing best practice requires to provision power based on the theoretical maximum power draw of the system (also called Worst-Case Provisioning (WCP)), despite the fact that only a synthetic workload comes close to this level of power consumption [Pat15]. One of the key contributions in the power-constrained domain is "hardware overprovisioning" [Pat13; Sar13a]. The idea is to provision much less power per node and thus provision more nodes. The benefit is that all of the scarce resource (power) will be used. The drawback is that power must be carefully scheduled within the system in order to approach optimal performance.

Fig. 2.3 depicts this foundational idea. Let the hardware overprovisioned system consist of N_{max} nodes and let the power budgeted for this system be P_{sys} Watts. As shown in the figure, with P_{sys} Watts total system power, only a subset of nodes (say N_{alloc}, where $N_{alloc} < N_{max}$ nodes) can be utilized at peak power (collection of nodes in red). Another valid configuration is to utilize the entire system (all the nodes) at low power. One of the several other intermediate configurations is to use medium power levels and utilize a portion of the system larger than that at peak power but smaller than that at low power. In each of these configurations, the system's power budget is uniformly distributed across varying number of nodes, i.e, each node is allocated $\frac{P_{sys}}{N_{alloc}}$ Watts of power. This is a naïve approach of enforcing a system power budget. Depending on the application's characteristics (memory-, compute-, and communication-boundedness), different applications achieve optimal performance on different configurations. In a nutshell, power procured for a system must be managed as a malleable resource to maximize performance of an overprovisioned system under a power constraint.

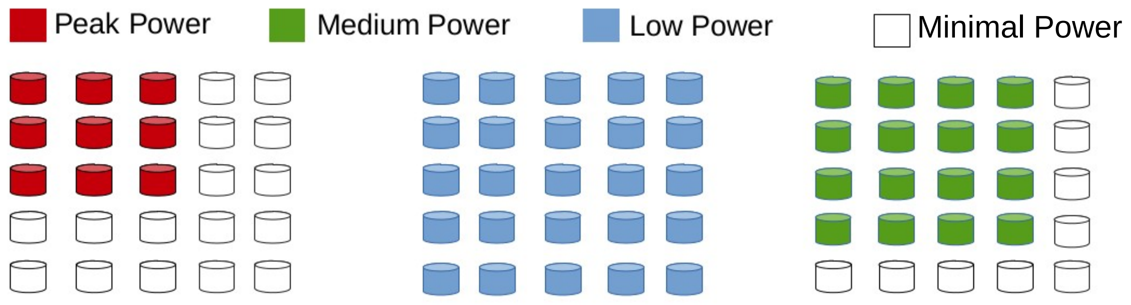


Figure 2.3 Hardware Overprovisioning under Power Constraint

2.3 Power control and measurement

As stated in Section 2.1.1, a compute node consists of multiple hardware components, each of which consumes power contributing to its operational power budget. Processors on the node are the main contributors to the node's power budget. The power consumption of a node can be controller by constraining the power of its processors.

2.3.1 Intel's Running Average Power Limit (RAPL)

From Sandy Bridge processors onward, Intel provides the Running Average Power Limiting (RAPL) [Int11] interfaces that allow a programmer to bound the power consumption of a processor, also called package (PKG) or socket. Here, package is a single multi-core Intel processor. Bounding the power consumption of a processor is called "power capping". RAPL also supports "power metering" to provide energy consumption information.

To set a power cap, RAPL provides a Model Specific Register (MSR), MSR_PKG_POWER_LIMIT. A power cap is specified in terms of average power usage (Watts) over a time window. Once a power cap is written to the MSR, a RAPL algorithm implemented in hardware enforces it.

For power metering purposes, RAPL provides MSR_PKG_ENERGY_STATUS and MSR_DRAM_ENERGY_STATUS registers. MSR_PKG_ENERGY_STATUS is a hardware counter that aggregates the amount of energy consumed by the package since the last time the register was cleared. MSR_DRAM_ENERGY_STATUS is a hardware counter that aggregates the amount of energy consumed by DRAM since the last time the register was cleared.

2.3.2 Dynamic Voltage Frequency Scaling (DVFS)

The power consumption of a processor (P_{proc}) can be divided into three main constituents, viz. dynamic power (P_{dyn}), short-circuit power (P_{sc}) , and static power (P_{static}):

$P_{proc} = P_{dyn} + P_{static} + P_{sc}$.

Dynamic power consumption is attributed to the charging and discharging of capacitors to toggle the logic gates during the instances of CPU activity. When logic gates toggle, there could be a momentary short circuit from source to ground resulting in short-circuit power dissipation. However, this loss is negligible [**kim2003leakage**]. Static power consumption is attributed to the flow of leakage current.

In today's systems, dynamic power is the main contributor to the power consumption of a processor [Mud01]. Dynamic power is proportional to the frequency of the processor, f, the activity factor, A, the capacitance, C, and the square of the supply voltage, V_{DD}. Dynamic power consumption of a processor can be managed by controlling its voltage and frequency. This is called voltage and frequency scaling:

$P_{dyn} = ACV_{DD}^2 f$.

Processor manufacturers have provided registers that can be used to configure the frequency of the processor [Int11]. Software can control the power consumption of a processor by dynamically modulating the frequency of the processor. This is called Dynamic Frequency and Voltage Scaling (DVFS).

Power measurements at node component-level granularity (e.g., processors, memory, GPU, hard-disk, fans) can be obtained via power acquisition systems such as Power Pack [Ge10] and Power Insight [III13]. A typical HPC server node is powered by Advanced Technology eXtended (ATX) power supply units (PSUs). The ATX PSU has three main voltage rails, 3.3V, 5V, and 12V, powering various node components. Power Pack and Power Insight both intercept individual power rails connected to the relevant node components and measure the current draw using shunts and Hall Effect sensors, respectively. Power is then calculated as the product of voltage and current. Node power can be measured using external power meters such as a Wattsup meter [Wat].

CHAPTER

3

POWER TUNER - POWER PARTITIONER : A VARIATION-AWARE POWER SCHEDULER

Research focus has only recently shifted from just performance to minimizing energy usage of supercomputers. Considering the US DOE mandate for a power constraint per exascale site, efforts need to be directed towards using all of the limited amount of power intelligently to maximize performance under this constraint.

3.1 Manufacturing variability and its impact on performance

As stated previously, uniform power capping is the naïve approach of enforcing a power constraint. In order to understand what happens under such a scheme, we characterized the performance of 600 Ivy Bridge processors on a cluster. We ran three of the NAS Parallel Benchmark (NPB) suite codes [Bai91], viz., Embarrassingly Parallel (EP), Block Tri-diagonal solver (BT), Scalar Penta-tridiagonal solver (SP), and CoMD, a molecular dynamics proxy application from the Mantevo suite [San11] at several different processor power bounds on all the processors. The processor power bounds were set using RAPL. The results are depicted in Figure 3.1. The x-axis represents operating power in Watts while the y-axis represents Instructions Retired per Second

(IPS) in billions. A maximum performance of 77, 50, 80, and 60 billions IPS is achieved for CoMD, EP, BT and SP, respectively. The cluster becomes non-uniform under power bounds with performance variations of up to 30% across this cluster for these applications. The potential causes of variability are discussed next but are effectively irrelevant as our proposed methods are agnostic of specific causes. More significantly, our experiments will show that this variability in performance translates into variation in peak *power efficiency* of the processors, which we exploit.

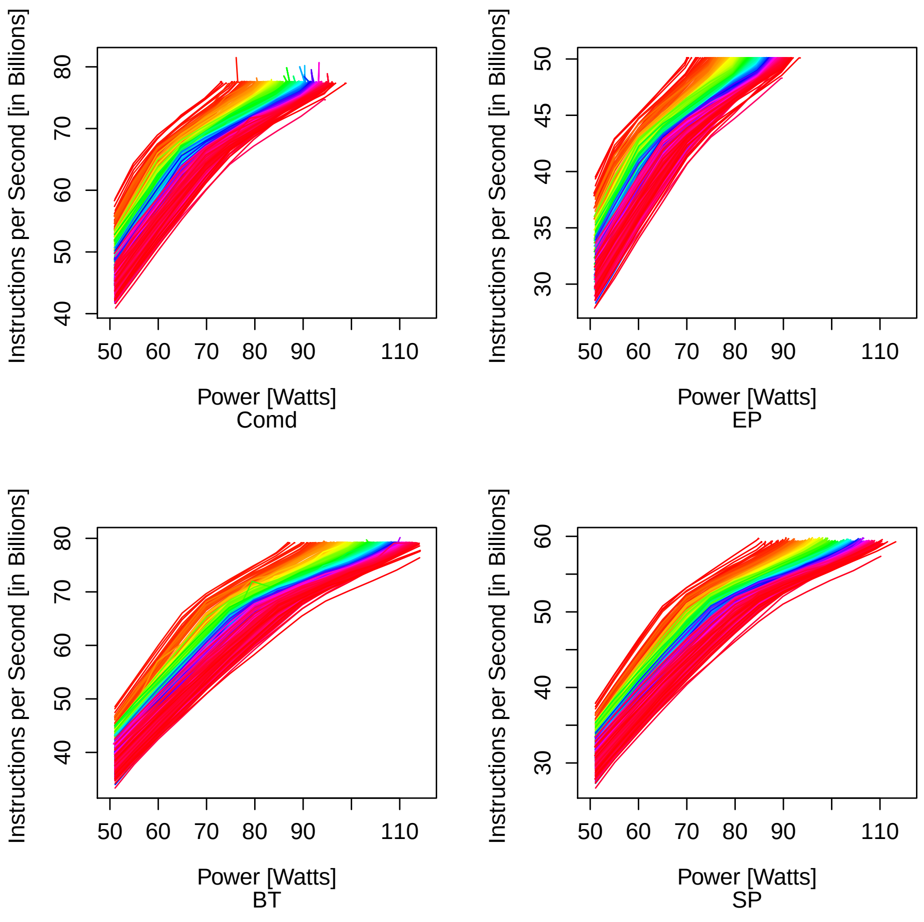


Figure 3.1 IPS vs. Power for each processor. Each rainbow line represent one processor. Curves in red (bottom) are least efficient, curves in orange (top) are most efficient.

Power Efficiency

Let *power efficiency* be defined as the number of instructions retired per second per Watt of operating power. Figure 3.2 represents the power efficiency curves of the processors on the cluster for the same set of codes. The x-axis represents the operating power in Watts and the y-axis represents the power efficiency in billion IPS/W. The rainbow palette represents different processors, where each curve (or each color) in the plots corresponds to a unique processor.

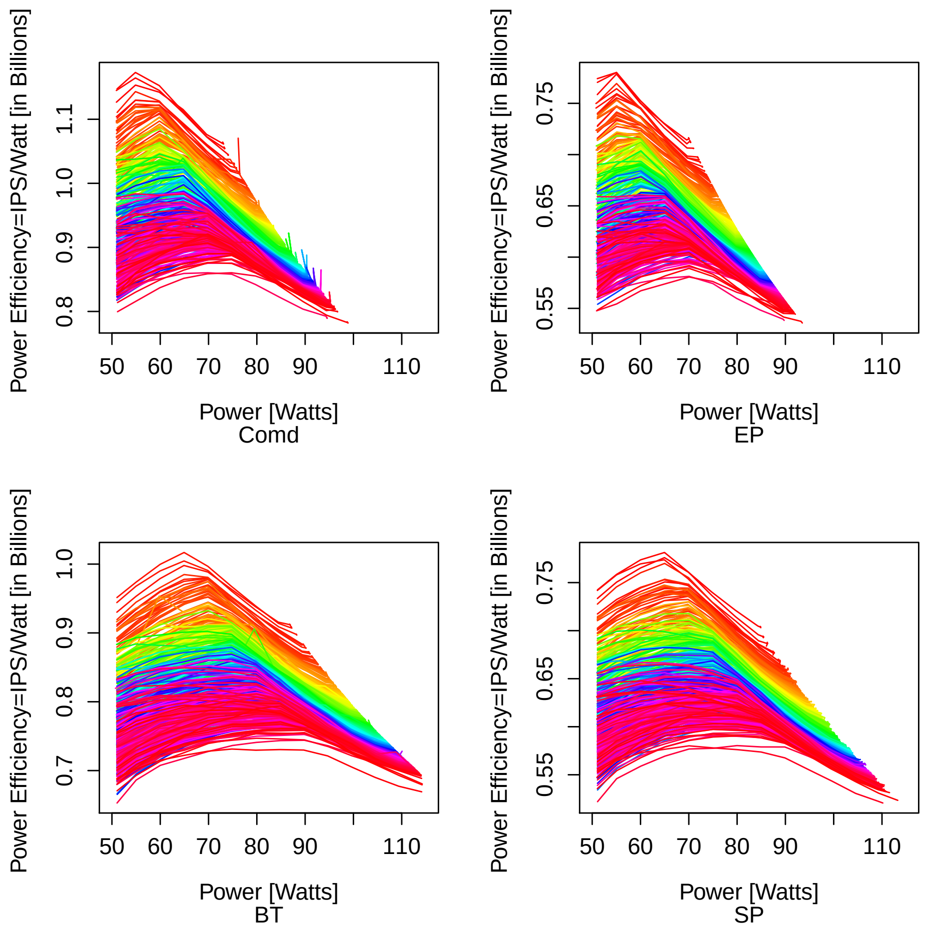


Figure 3.2 Power Efficiency in IPS/W vs. Operating power. One rainbow line per processor, red curves (bottom) are least, orange ones (top) most efficient.

We make the following observations from these experiments:

- The power efficiency of a processor varies with its operating power and is non-monotonic. It is also workload-dependent.
- Peak power efficiency varies across processors.
- Most importantly, efficient processors are most efficient at lower power bounds whereas the

inefficient processors are most efficient at higher power bounds. The "peak" of every curve is the point at which the processor achieves the maximum efficiency, i.e., maximum IPS/W. Orange curves (efficient processors) have peaks at lower power compared to the peaks of the red curves (less efficient processors) and the rest lie in between.

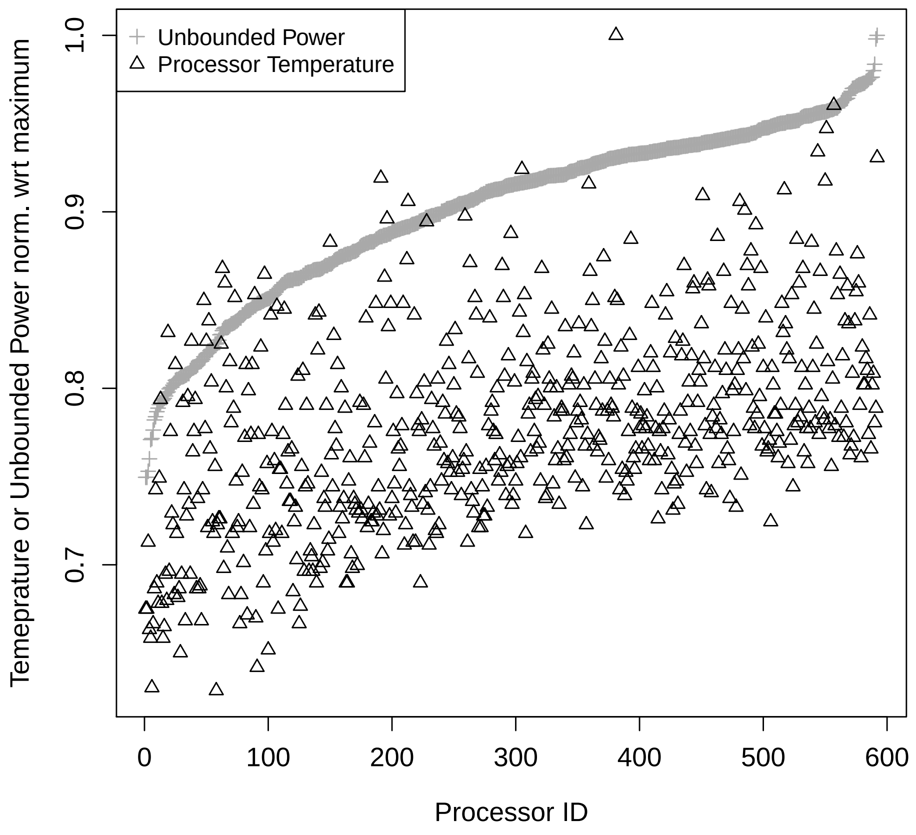


Figure 3.3 Temperature and unbounded power of processors. Processors are sorted by unbounded power consumption.

Figure 3.3 depicts the results of our thermal experiments. The x-axis presents processor IDs (processors are sorted in the order of efficiency). The y-axis presents the measured temperature (triangles) of the processors normalized with respect to the maximum temperature and the unbounded power (crosses) of the processors also normalized with respect to the maximum power. In these experiments, the processors were not capped, and they achieved uniform performance. We observe that the temperature increases as we go from efficient to inefficient processors (left to right), as does the unbounded power. However, not all inefficient processors are hotter than the efficient ones. This shows that thermal variation may be one of the potential causes of variation in efficiency but there are other factors that counter the effect as we do not see a linear trend for temperature (in contrast to the linear trend of unbounded power). We believe that one of the contributing factors is process/manufacturing variation induced at the time of fabrication. In the end, our proposed mechanism is agnostic of the actual cause of variation, it simply exploits the fact that variation (due to whatever reason) exists.

In summary, there exists variation in power efficiency across processors. There is a unique local maximum in every power efficiency curve that occurs at disparate power levels for different processors. Starting from the minimum power, increasing the power assigned to a processor leads to increasing gains in IPS. However, increasing the power beyond the peak efficiency point of a processor leads to diminishing returns. Hence, when power is limited, processors should operate at power levels close to their peak efficiency to maximize the overall efficiency of the system. Since the peak efficiency points for efficient processors are at lower power levels than for the inefficient processors, the optimal configuration should select lower power levels for efficient processors and higher power levels for inefficient processors to maximize performance. On the contrary, a naïve / *uniform power* scheme caps all the processors at identical power bounds. Hence, it is sub-optimal.

An optimal algorithm should aim at leveraging the non-uniformity of the cluster to maximize the performance of a job under its power constraint.

To this end, we propose *PTune*, a power-performance variation-aware power tuner that exactly does this for each job. For every job, given a power budget, it determines the following: (1) the optimal number of processors (say n_{opt}); (2) selection of n_{opt} processors; and (3) the power distribution (say p_k, where $1 \leq k \leq n_{opt}$) across the selected n_{opt} processors.

The problem statement can be stated as follows: Given a machine level power budget, how should the machine's power be distributed across (a) jobs and (b) tasks within jobs on a given system, where (b) is discussed later. For (a), the process of making these decisions at the macro level of jobs is called *power partitioning*. Each job on the machine receives its own power partition.

We address the following questions:

1. How many partitions do we need at a time? I.e., determine how many jobs should be scheduled at a time.

2. What is the size of each of the power partitions? I.e., determine the power budget assigned to each job.

For (b), at the micro level, given a hard job-level power budget P_{Ji}, we need to determine the optimal number of processors, n_{opt}, with a power distribution (p_1, p_2, ... , $p_{(n_{opt}-1)}$, $p_{n_{opt}}$) such that performance of the job is maximized under its power budget. The constraint on the power distribution is expressed as

$\sum_{k=1}^{n} p_k \leq P_{Ji}; min\_power \leq p_k \leq max\_power_k.$

Here, min_power is the minimum power that needs to be assigned to a processor for reliable performance and max_power_k is the maximum power consumed by the k^{th} processor (uncapped power consumption) for an application. The performance of a job can be quantified in terms of number of instructions retired per second (IPS). The general model holds for other performance metrics as well. We selected IPS here because it closely correlates to power in our experiments. For a parallel application on n processors, the effective IPS is the aggregated IPS over n processors ($JobIPS_n$). Hence, the objective function is

$Maximize(JobIPS_n)$.

A processor's IPS is a non-linear function of the power at which it operates. Each processor can be power bounded at several levels using the RAPL capping capabilities, which forces it to operate at various power levels within a fixed range. We know that unbounded power consumption is variable across processors while achieving the same unbounded (peak) performance for a given application. This is depicted in Figure 3.4. The x-axis indicates the power at which the processor operates and the y-axis shows the IPS (in billions) of the processor of an application. Each solid curve corresponds to the most efficient processor while the dotted curve correspond to the least efficient processor.

The following two observations are made from this data:

1. On a single processor, the performance (IPS) achieved at any fixed power level is different for different workloads.
2. The performance of an application on two different processors at any fixed power level is not the same.

This means that when determining the optimal distribution of power across processors it is necessary to take the processor characteristics and the application characteristics into account. One solution may not fit all applications. The optimal configuration for an application on one set of processors may be different from that on another set of processors because of performance variations under a power cap.

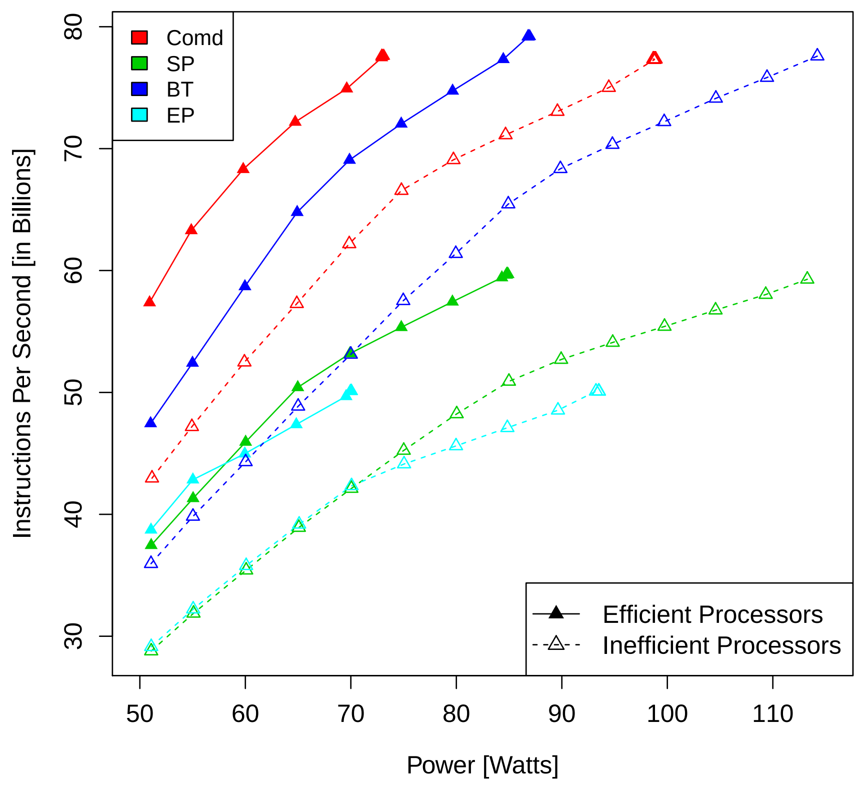


Figure 3.4 IPS vs. Power for efficient and inefficient processors

To target the sub-optimal throughput problem we propose a 2-level hierarchical approach of managing power as a resource (see Figure 3.5). The parameters of the model are described in Table 3.1. N_{max}, $P_{m/c}$ and n_{req} are the inputs to the model that we assume. n_{opt} is calculated once for every job at its dispatch time. N_{alloc}, P_{Ji} and p_k are re-calculated every time any job is dispatched. min_power is architecturally defined for every family of processors. Table 3.2 is populated off-line using the characterization data. We make the assumption that the power consumption of the interconnect is zero, i.e., interconnect power is beyond the scope, and so are task-to-node mapping effects on power. We only consider processor power in this work and assume moldable jobs. DRAM power could not be included due to motherboard limitations at the time of this work. We do not expect the users of the system to predict and request power in their job request. Power decisions are made by our system software (PTune and PPartition). Users may be allowed to influence these decisions by assigning priorities to their jobs.

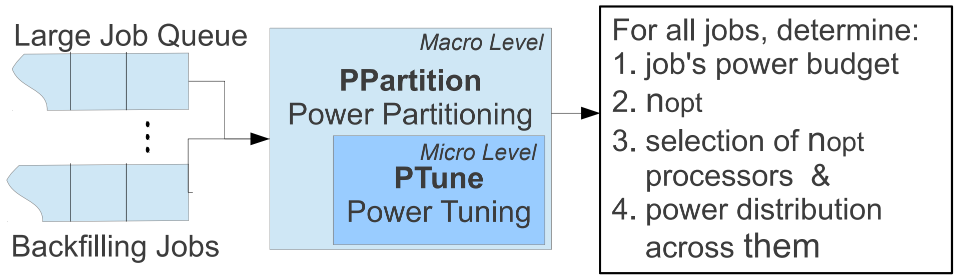


Figure 3.5 Hierarchical Power Manager

Table 3.1 Model Parameters

Parameter	Description	PPartition	PTune
N_{max}	maximum number of processors on a machine	Input	N/A
N_{alloc}	number of processors already allocated to jobs	Output	N/A
$P_{m/c}$	power budget of the machine	Input	N/A
P_{m/c_unused}	unused power budget of the machine	Variable	N/A
n_{req}	number of processors requested by a job	Input	Input
n_{opt}	optimal number of processors for a job	Output	Output
n	number of processors for a job under its power budget	N/A	Variable
P_{Ji}	power budget of the i^{th} job	Output	Input
p_k	power cap of k^{th} processor within a job	N/A	Output
min_power	minimum processor power cap	Input	Input
max_power_j	maximum processor power cap of the j^{th} processors	Input	Input
power-ips table	characterization data Table 3.2	Input	Input

Table 3.2 power-ips lookup table (last metric in Tab. 3.1)

Power Cap [W]	IPS in Billions	Measure Power [W]
60	46.43	59.99
80	64.83	79.88
100	76.33	99.43
120	79.13	104.66

At the macro level, we propose *PPartition*, a technique of partitioning a machine's power budget across jobs while scheduling them. Once a job is dispatched by a conventional scheduler (e.g., slurm or Maui/pbs), PPartition calculates its power budget. If the required power is not available, it steals power from the previously scheduled jobs and provisions this power for the new job. If sufficient power cannot be obtained, PPartition overrides the conventional scheduler's decision based on free resources (nodes) and does not schedule this job until sufficient power is available.

At the micro-level, we propose *PTune*, a power balancing model that determines the distribution of a job's power budget (one job at a time) across an optimal selection of processors (among all free resources) to maximize the performance of a job under its power budget.

PTune shrinks the job's processor allocation by eliminating the less efficient processors that are expensive in terms of power to maximize the performance of a job under its power budget. Figure 3.6 depicts the micro-level power tuner. For each job Ji, a power budget P_{Ji} is calculated at the macro level by PPartition. For every job with this assigned power budget, PTune answers the following questions:

1. How many (n_{opt}) and which processors should a job run on?
2. What should be the power ($p_1, ..., p_{n_{opt}}$) assigned to each of the n_{opt} processors?

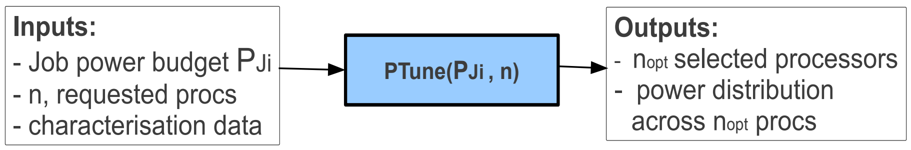


Figure 3.6 PTune

3.2 PTune

Let us start by addressing the first question. In order to use a processor, it needs to be assigned at least the minimum power (min_power) that is constant across all processors. The upper limit on the processor's power (max_power_k) is variable across processors.

Figure 3.7 shows the maximum power consumption of 600 Ivy Bridge processors when they are not power capped. The unbounded performance is uniform across all the processors. The x-axis represents all the processor sorted by power consumption and the y-axis represents the maximum power consumption in Watts. The optimal configuration for maximum performance of a job under a strict power budget consists of the maximum number of most efficient processors (from the left) such that their aggregate power consumption does not exceed the job's power budget.

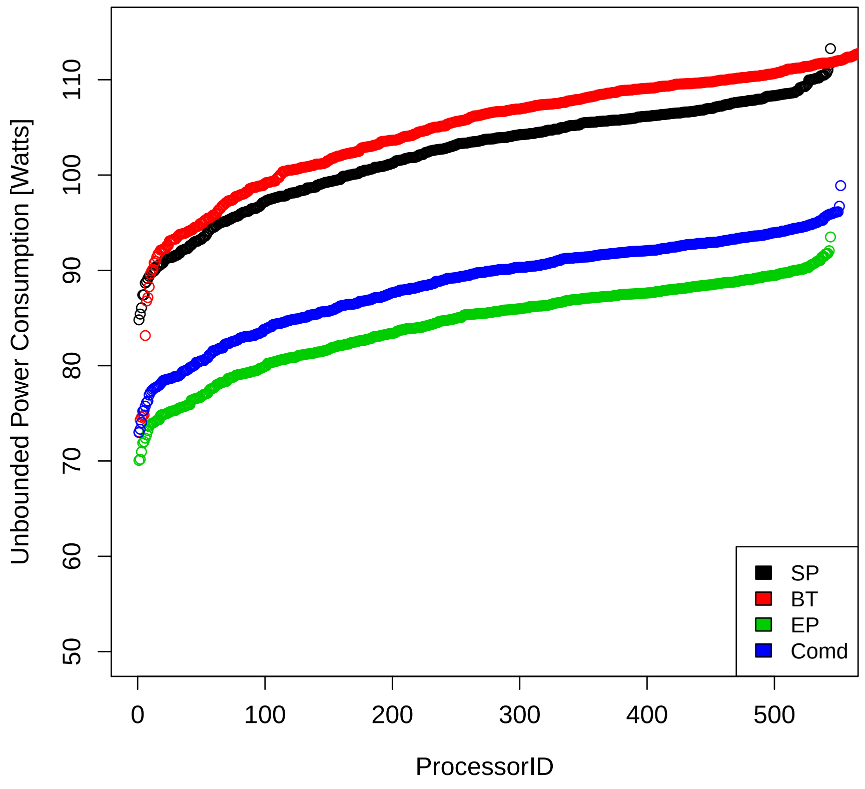


Figure 3.7 Unbounded power consumption of processors under uniform performance

3.2.1 Sort the Processors

The first step towards determining the optimal configuration is to sort the available processors by their relative power efficiency. This is equivalent to sorting them by their unbounded power consumption. Let the sorted set of processors be indexed by k.

We divide this distribution of processors into quartiles, viz., Q1, Q2, Q3 and Q4, in the order of efficiency and pick processors from one or more of these quartiles for evaluation purposes.

3.2.2 Bounds on Number of Processors

The lower bound on n, $n_\perp$, can be calculated by determining the maximum number of processors that can be capped at their maximum power, max_power_k, under the power budget. The selection of processors is reformed in the sorted order as described above. $n_\perp$ is given by the largest value of n that satisfies the following constraint:

$$P_{Ji} \geq \sum_{k=1}^{n} max\_power_k.$$

The upper bound on n, $n_\top$, represents the maximum number of processors that can be operated at min_power under the power budget. The bound $n_\top$ is calculated as follows:

$$n_\top = \frac{P_{Ji}}{min\_power}.$$

The processor count, n, is iterated from $n_\perp$ to $n_\top$, and in each step, the next efficient processor is added to the set of processors. Job-level performance, $JobIPS_n$, is calculated in each iteration by $DistributePower()$ for the power budget P_{Ji} and a given number of processors, n, where $n_\perp \leq n \leq n_\top$.

The optimal number of processors, i.e., n_{opt}, is the value of n at which a job's IPS is maximized.

$$JobIPS_{n_{opt}} = max(JobIPS_{n_\perp}, JobIPS_{(n_\perp+1)}, ..., JobIPS_{n_\top}).$$

PTune leads to $n_{opt} \leq n$. Thus, PTune tends to reduce the number of processors required for a moldable job The spare processors are returned back to the global pool of unused resources so that they can be utilized by other jobs.

3.2.3 Distribute Power: Mathematical Model

DistributePower(), takes three inputs, viz., the number of processors n, the job's power budget, P_{Ji}, and the power distribution across $n-1$ processors determined in the previous iteration. The output of this function is the maximum job IPS that can be achieved under P_{Ji} Watts with n processors. It also calculates the optimal power caps, $(p_1, ..., p_n)$, for n processors, which forms an input for the next iteration. This can be mathematically expressed as follows:

$$\begin{aligned} &DistributePower(n, P_{Ji}, (p_1, ..., p_n)) = \\ &\quad DistributePower((n-1), P_{Ji} - p_n, p_1, ..., p_{(n-1)}) + \\ &\qquad\qquad\qquad getProcIPS(n, p_n). \end{aligned}$$

The function getProcIPS(k,p_k) performs a look-up in Tab. 3.2 to return the expected performance (IPS) of the k^{th} processor when it is capped at p_k Watts.

3.2.4 Power Stealing and Shifting

DistributePower() consists of two main steps, viz. Power Stealing and Power Shifting.

Step 1: Power is stolen in discrete quantities ($delta\_power$) from the $n-1$ processors to provision power for the n^{th} processor (see Figure 3.8). The victim/donor processor is the one that suffers minimum loss in IPS when $delta\_power$ is stolen from it. If the aggregate stolen power is at least min_power, an additional n^{th} processor is added to the processor set.

Step 2: Power is shifted from a donor to a receiver in discrete quantities, $delta\_power$, across the n processors. The victim/donor processor is identified in the same way as in step 1. The receiver is the processor that gains maximum IPS on receiving $delta\_power$.

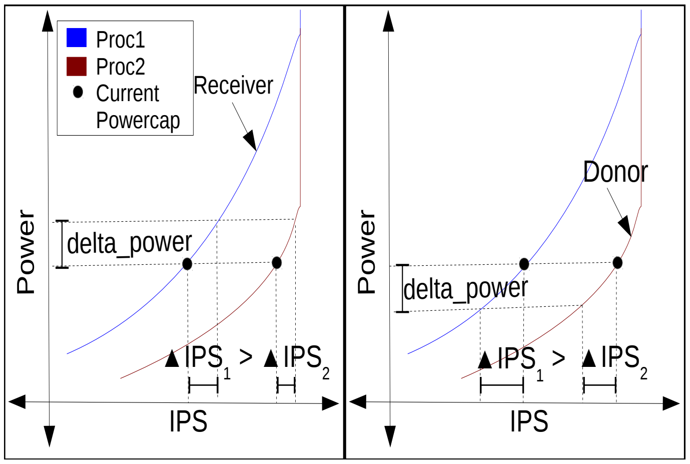


Figure 3.8 Donor and receiver of discrete power

3.3 PPartition

Figure 3.9 depicts the macro-level power partitioning algorithm. The power partitioner co-operates with the conventional scheduler (simulated in R). PPartition receives information on the performance variations across processors. It always chooses the most efficient n_{req} processors of the available processors (or a subset thereof) to schedule a job. The conventional scheduler dispatches a job from the job queue when the requested number of processing resources are available. When job J_i is dispatched, its initial power budget, P_{Ji}, is calculated as follows:

$P_{Ji} \leftarrow P_{m/c} * \frac{n_{req}}{N_{max}}$

If the required power is available, PTune determines the optimal configuration for the job and the job is scheduled. It is important to note that even though the job power budget is proportionate to the number of requested processors, PTune schedules jobs on reduced number of processors. As a result, the machine's power depletes at a faster rate that the processing resources. If the available (unused) power is less than the calculated job power budget, power is stolen from already scheduled jobs. This is called power repartitioning (lower right blue/shaded box in Figure 3.9) and detailed next.

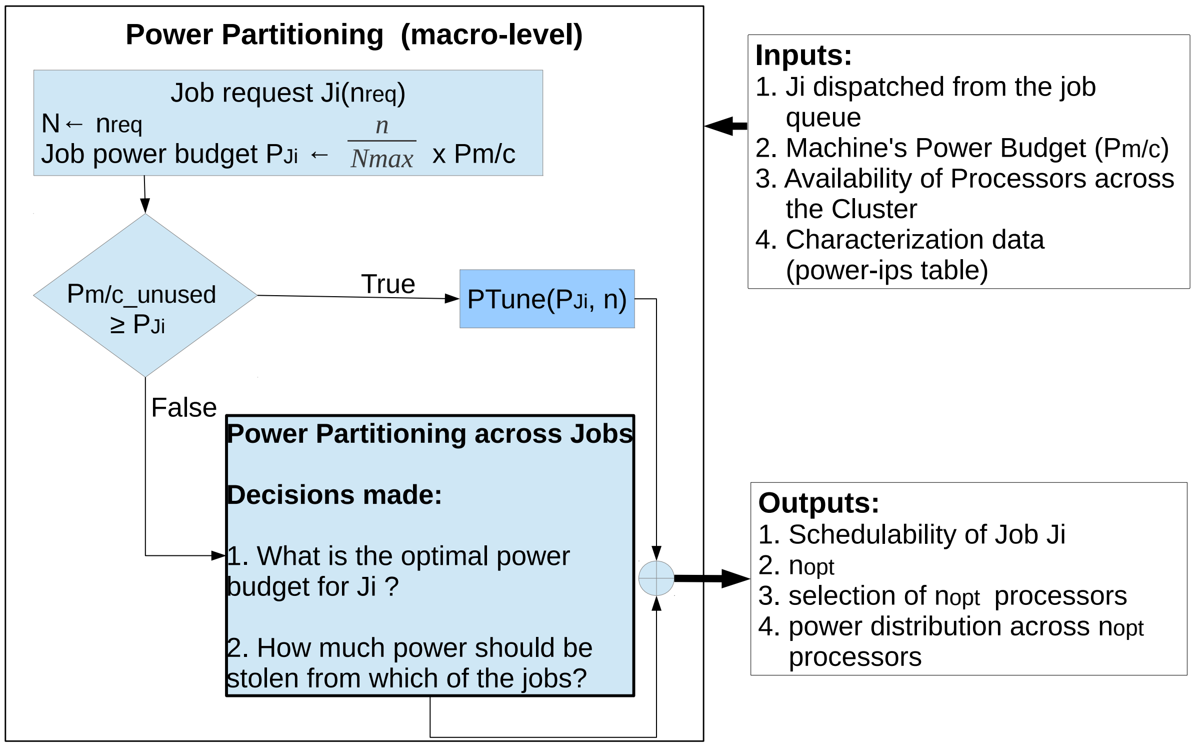

Power Partitioning (macro-level)
Job request Ji(nreq)
N ← nreq
Job power budget PJi ← n / Nmax x Pm/c
Pm/c_unused ≥ PJi
True
PTune(PJi, n)
False
Power Partitioning across Jobs
Decisions made:
1. What is the optimal power budget for Ji ?
2. How much power should be stolen from which of the jobs?
Inputs:
1. Ji dispatched from the job queue
2. Machine's Power Budget (Pm/c)
3. Availability of Processors across the Cluster
4. Characterization data (power-ips table)
Outputs:
1. Schedulability of Job Ji
2. nopt
3. selection of nopt processors
4. power distribution across nopt processors

Figure 3.9 PPartitioning: Repartitioning Power

Power Repartitioning

The power repartitioning algorithm is shown in Algorithm 1. As all of the machine power budget is already used up by the $N_{allocated}$ processors, a fair power share for the new job is calculated as

$P_{Ji} = P_{m/c} * \frac{n}{n+N_{allocated}}$, where n=$n_{req}$.

The job is power tuned for the requested n_{req} (assigned to n) processors under P_{Ji} Watts

Algorithm 1 Repartitioning Power For Accommodating the i^{th} job

```
1: procedure PowerPartitioner($J_i$,$n_{req}$)
2:     $n_{opt} \leftarrow n_{req}$
3:     $n \leftarrow n_{opt}$
4:     $P_{Ji} \leftarrow P_{m/c} * \frac{n}{n+N_{allocated}}$    ▷ Recompute $P_{Ji}$ proportional to the portion of busy processors requested
5:     $n_{opt} \leftarrow PTune(P_{Ji}, n)$    ▷ Recompute $n_{opt}$ $n_{opt} < n$
                                             ▷ Repartition power across jobs to provision power for the $i^{th}$ job
6:     for $k \leftarrow 1; k < i; k++$ do
7:         $power\_to\_be\_stolen[k] \leftarrow (P_{Ji} - P_{m/c\_unused}) * \frac{jobpowerbudgets[k]}{\sum jobpowerbudgets}$
8:         $total\_stolen\_power \leftarrow total\_stolen\_power$
                   $+ ShrinkPartition(power\_to\_be\_stolen[k], k)$
9:     end for
10:    if $total\_stolen\_power < P_{Ji}$ then ▷ If enough power cannot be stolen, recompute $n_{opt}$
11:        $P_{Ji} \leftarrow total\_stolen\_power$
12:        $n_{opt} \leftarrow PTune(P_{Ji}, n)$
13:    end if
14: end procedure
```

calculated above. PTune gets rid of the unaffordable less efficient processors, if any, leading to $n_{opt} \leq n$. We recompute (in the while loop) the proportionate power the new job with n_{opt} processors should have due to power partitioning across all jobs. This new power budget, P_{Ji}, then becomes the base for another PTune, and so on, until the number of processors (monotonically decreasing) for the new job reaches a fixed point (stabilizes) in the while loop. The fixed point guarantees a fair power level (P_{Ji}) relative to other jobs, but we still need to find other jobs to steal just enough power for this job.

In the following for loop, power is stolen from each of the scheduled jobs in a proportionate manner to each other's power budget. This is accomplished by *ShrinkPartition*, which consists of (1) stealing just enough power and (2) power tuning for the remaining power of a job and the same number of processors (since we assume moldable but not malleable applications). Here, we steal as much power as possible while retaining heterogeneous power bounds across a job's processors to respect processor variations and thus ensure a high IPS under lower power budget.

The aggregate stolen power from other jobs is offered to the new job. If the stolen power is less than the fixed power level for the new job, which was P_{Ji}, then the new job needs to be tuned one more time. If the stolen power was sufficient for this last tuning step, the new job is scheduled and the power re-tuning decisions made by ShrinkPartition for the existing ones are enforced. If, however, the stolen power is insufficient (as determined by PTune when the power budget cannot accommodate more than $\frac{n}{2}$ processors), no power is redistributed, i.e., all jobs remain unchanged in their power settings and the new job is deferred until at least another job completes.

3.4 Implementation and Experimental Framework

We modified the libmsr [Sho14] library to gather the processor characterization data. We implemented a power-performance profiler using the MPI profiling interface (PMPI) that invoked various subroutines of the libmsr library to assess the power and the performance of MPI applications. We captured several fixed counter values, power consumption, and completion times for each application on all the processors. The processor power consumption was measured using Intel's RAPL interface. This characterization data is made available to PTune and PPartition.

We assume that the jobs are moldable. Our power manager works in co-ordination with the conventional job scheduler. Once a job is dispatched by the conventional scheduler, the power manager (PPartition+PTune) determines its power budget, the selection of processors from those available, and the power distribution (or processor power caps) across them.

We assume a large job queue (> 384 processes) and a backfilling queue (< 48 processes). The conventional job scheduler schedules as many large jobs as it can on the machine before scheduling the backfilling jobs. We assume up to Nmax=550 nodes with 12 cores each (6600 processes). If the power manager decides to schedule the job, power distribution across its processors (and power repartitioning if required) is enforced using RAPL.

Experiments were conducted on a 324-node Ivy Bridge cluster. Each node has two 12-core Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz processors and 128 GB of memory. We used MVAPICH2 version 1.7. The codes were compiled with the Intel compiler version 12.1. The msr-safe kernel module provides direct access to Intel RAPL registers via libmsr [Sho14]. We used the package (PKG) domain of RAPL that provided us the capability of capping power for each of the processors in an experiment. The scheduling environment was simulated in R.

We again used EP, BT, and SP from the NPB suite and CoMD from the Mantevo suite in their pure MPI versions. We exponentially increase the node count for our experiments. The inputs were weakly scaled for different node counts. We report performance in terms of completion time in seconds and power in Watt. The reported numbers are averages across ten runs.

3.5 Experimental Evaluation

Experiments were conducted for single job power tuning and multi-job power partitioning.

Variation under Power Caps: Sorting Required

We now exploit the observed variability in the unbounded power consumption of the processor chips, which translates into variation in performance under a power constraint. This variability may be caused by factors such as manufacturing/process variation (at CMOS/transistor level), ambient machine room temperature in different rack positions (higher/lower to the floor), or others. Yet, our method handles variation irrespective of its causes. Previous work [Rou12] and Section 5.1 has already established that a cluster is not homogeneous under a power constraint because of such variation. We also observe that scheduling a job on different sets of fixed number of processors under a constant power budget leads to variation in the performance of a parallel job.

We present a selection of configurations to demonstrate this behavior in Figures 3.10 and Fig. 3.11. The x-axis represents the codes and the number of processors. The y-axis indicates the completion time in seconds. The codes are run on several combinations of processors from one or more quartiles of the processor distribution. The numbers on the top of the bars indicate percentage slowdown with respect to the baseline. The processors are uniformly capped at 51W in this set of experiments, i.e., they maintain a constant job power budget of 8KW, 16KW, and 32KW for 16, 32, and 64 processor experiments, respectively. The baseline for 16 processor experiments (Figure 3.10) is the performance on the processors belonging to quartile Q1. For 32 and 64 processors (Figure 3.11), the baseline is the performance on the processors belonging to Q1 and Q4 (also see legends). Q1 consists of the most efficient processors whereas Q4 consists of the least efficient processors. We observe a performance slowdown ranging from 2% to 18%. We observe that performance deteriorates as we include less efficient processors (Q2, Q3, Q4) in the mix. Hence, the optimal selection of n_{opt} processors should consist of the most efficient processors from the available ones.

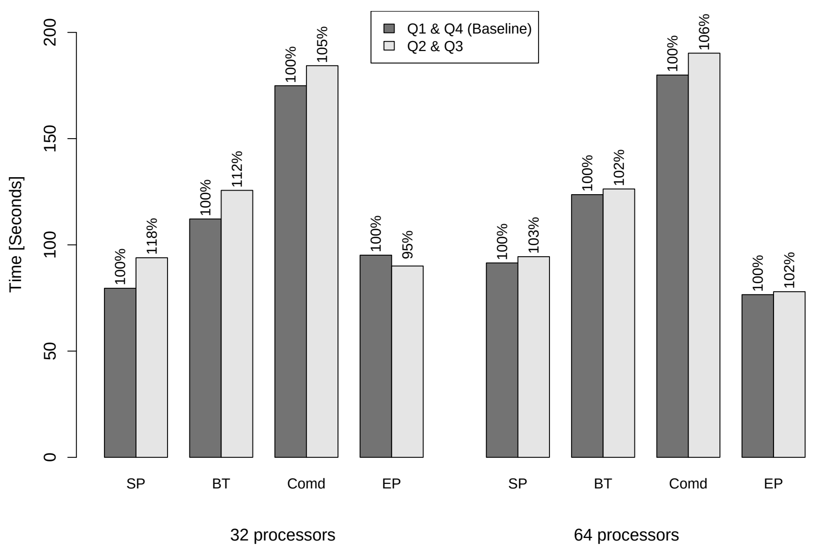


Figure 3.11 Performance variation on 32 and 64 processors

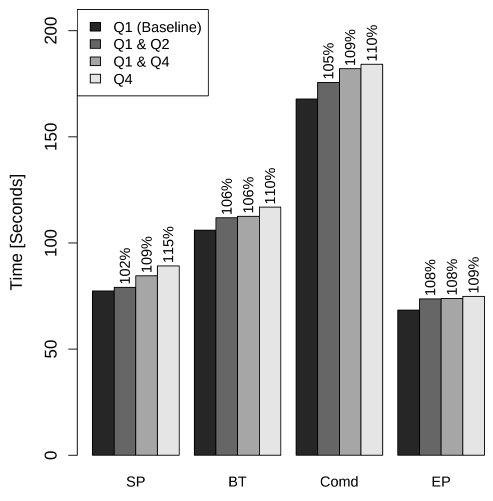


Figure 3.10 Performance variation on 16 processors

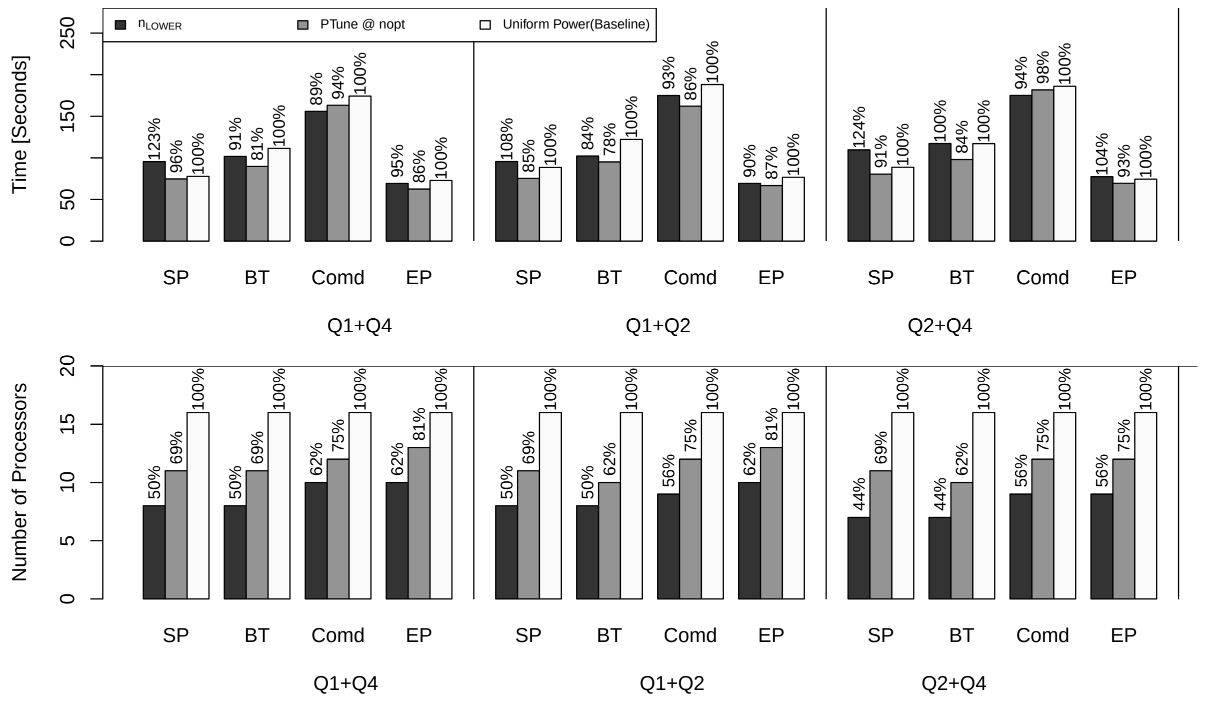


Figure 3.12 Evaluation of PTune on 16 processors from one or more quartiles

PTune

We evaluate the effectiveness of PTune using the aforementioned codes. In Figure 3.12, we present results for three different combinations of processors belonging to different quartiles. There are three data points corresponding to each code.

In the figures, n_{LOWER} (synonymous with $n_{\perp}$) is the maximum number of processors that can operate at maximum power such that their aggregate power does not violate the job level power constraint. This configuration most closely resembles the worst case power provisioning as processors are not power constrained. PTune is the data point corresponding to optimal configuration suggested by the power tuner. Uniform power corresponds to the naïve approach of distributing the job's power budget evenly across all processors in a job. This is the baseline configuration.

Performance

In Figure 3.12, the y-axis represents performance (top graph) in terms of wall-clock time (in seconds) and the number of processors recommended by the power manager (bottom graph) over different codes and quartiles to which the processors belong (x-axis). The numbers on the bars indicate the

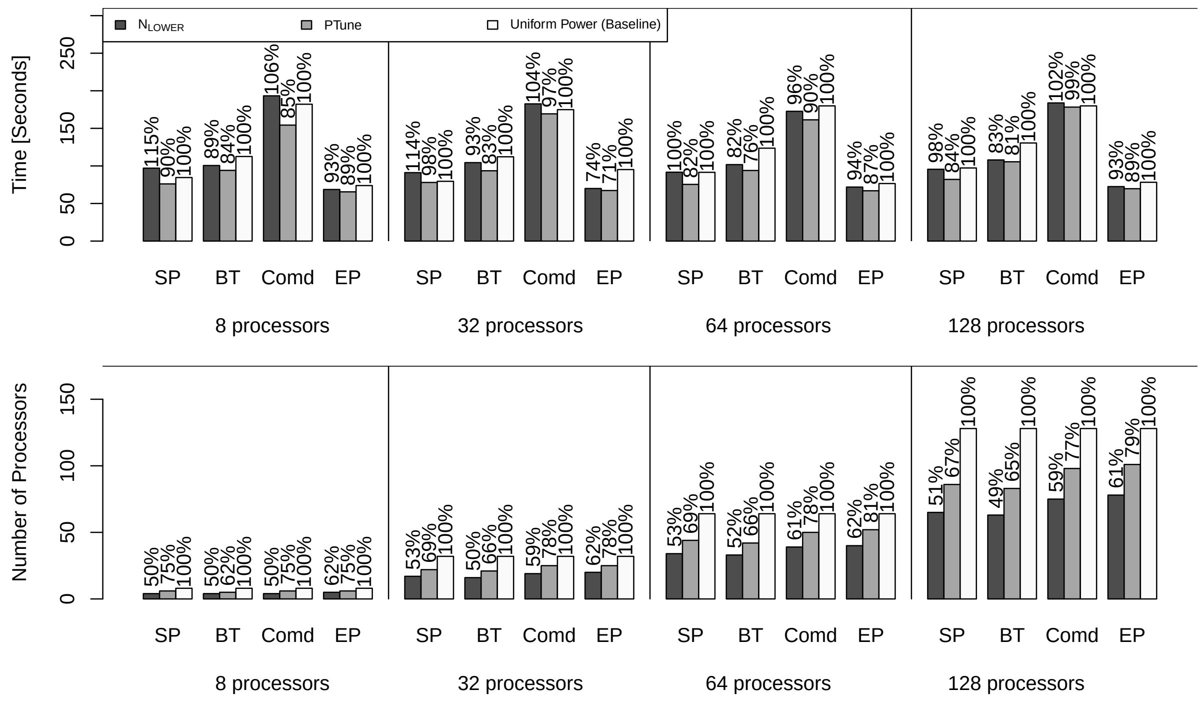


Figure 3.13 Evaluation of PTune on processors from Q1 and Q4 quartiles

runtime reduction and utilized number of processors relative to the baseline in percent.

We observe a performance improvement of up to 22%. The gains are dependent on the combination of processors from different quartiles as well as on workload. PTune is able to free up to 38% of the resources while achieving similar or higher performance than the baseline configuration.

Scalability

We evaluate PTune on up to 128 processors. Figure 3.13 presents results addressing the scalability of PTune. PTune achieves performance improvements of as much as 29% with a minimum of 1%. More significantly, in case of the minimal performance improvement, PTune frees up 23% of the processors, which subsequently become available to the next scheduled jobs. We observe an error of less than 2% between the total job power consumption (measured via RAPL) of the PTune recommended configurations and the assigned job-level power budget across all experiments.

PPartition

In this section, we perform a macro-level evaluation of our 2-level model. We simulate the conventional scheduler that dispatches jobs from multiple queues, one at a time. Let np be the number of processes. The scheduler handles 3 queues, 1 large job queue ($np \geq 768$ or $n \geq 64$ processors), and two backfill queues ($np \leq 48$ or $n \leq 4$ processors, $48 < np < 768$ or $4 < n < 64$ processors). Larger jobs are scheduled first followed by backfilling jobs to improve the system utilization. We assume $Nmax = 550$ processors. Our job mix consists of 25% jobs from each EP, SP, BT, and CoMD.

We assume a hardware overprovisioned machine with a machine power budget $P_{m/c} = 28KW$. Figure 3.14 depicts a scenario in which the job scheduler is oblivious of power management. The machine's power budget is uniformly distributed across all the processors. We call this naïve scheduling. The conventional scheduler schedules jobs as long as the required number of processors are available. Figure 3.15 depicts the scenario when our power manager (PTune + PPartition) co-ordinates with the conventional scheduler to make variation-aware power and job scheduling decisions. The x-axes in both the plots represent job identifiers ordered by the time that they are dispatched by the conventional scheduler. We can see that the large 64 processor job is scheduled first followed by the backfilling jobs. The left y-axis denotes job performance in IPS. Each of the red, green and blue curves represents a job's performance as more and more jobs are scheduled over time (moving right along the x-axis).

Our scheduler starts with jobs at high power budget and, hence, high performance. But as more jobs are dispatched, power is stolen from the previously scheduled jobs. This leads to a drop in their performance. In return, we are able to schedule more jobs at the expense of the performance of already running jobs. In this scenario, our scheme is able to schedule 58 jobs whereas the power-oblivious scheduler is able to schedule only 36 jobs to run at the same time. This is because PTune schedules each job on a reduced number of processors compared to the naïve scheme. The performance of most of the first 36 jobs (that are scheduled under both the schemes) of our approach is at least as good as the naïve one. In addition to these jobs, our power control is able to schedule 22 more backfill jobs that further improve the overall throughput (SysIPS) of the machine (compared to the naïve approach) under the same power constraint.

The right y-axes depict the system's power consumption as a fraction of (normalized to) the overall provisioned system power, $P_{m/c}$, in one line graph (circles) and the system's performance ($SysIPS = \sum JobIPS_i$) normalized to the maximum in the other (crosses). Both graphs track each other closely, but under our power control, the machine power is fully utilized much earlier (after ≈ 10 jobs) whereas 36 jobs are required to reach this level in the naïve case. These initial jobs also achieve higher performance under our scheme (>1500 Billions IPS) than that in the naïve case (900 to 1500 Billion IPS for backfill jobs and 2100 Billion IPS for the large job) before the other jobs are

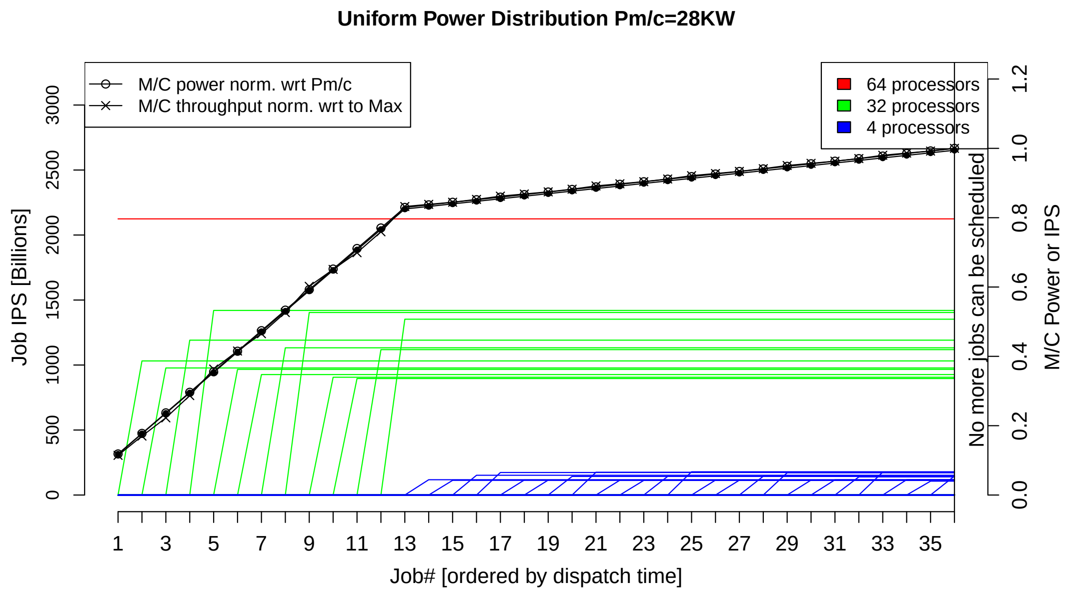


Figure 3.14 Uniform power distributed across the machine. $P_{m/c} = 28kW$

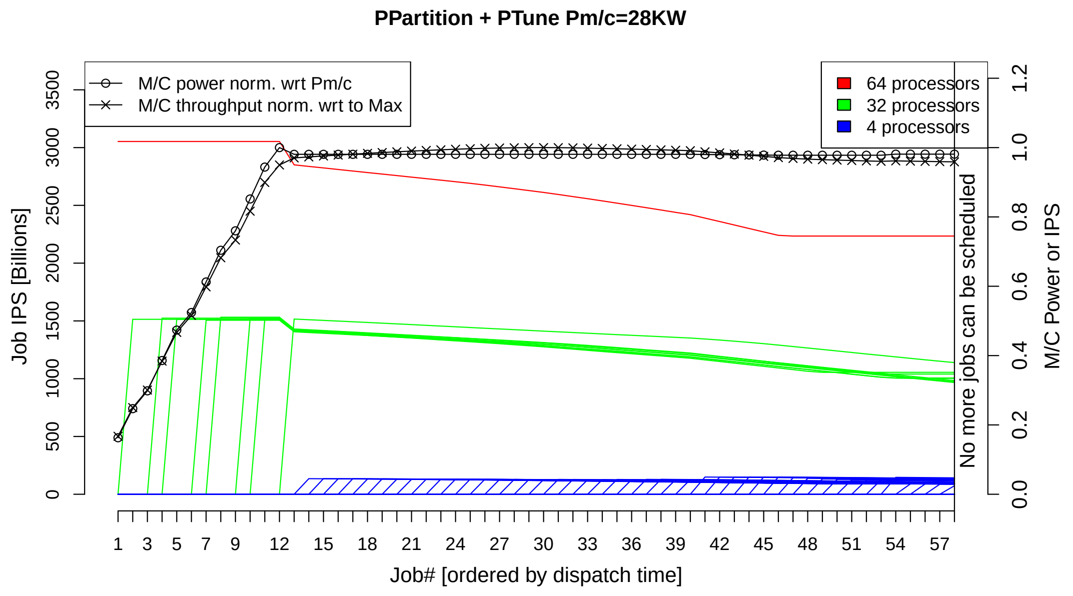


Figure 3.15 PPartition + PTune. $P_{m/c} = 28kW$.

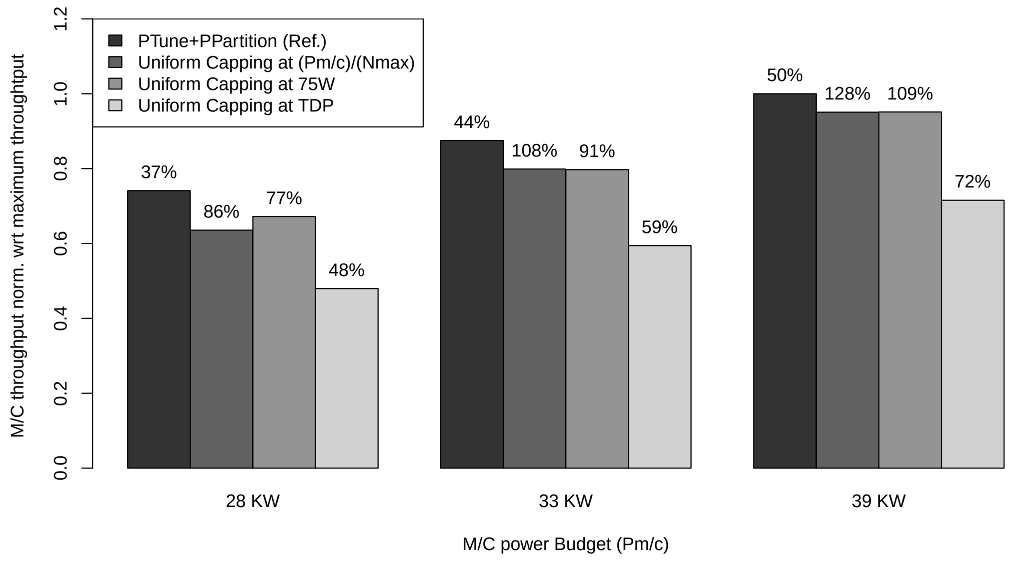


Figure 3.16 Throughput

scheduled, and these jobs would thus terminate earlier as they have progressed further under our power control compared to the naïve case. This shows that when there are fewer jobs running on a machine, our power manager is able to direct all machine power to jobs where it is needed to maximize performance under a power constraint unlike the naïve approach.

Figure 3.16 presents a comparison of the throughput of our scheme compared to three other naïve schemes. The x-axis denotes the machine's power budget and the y-axis depicts the throughput of the machine (SysIPS) normalized to a maximum throughput at 39KW (left bar per set). Uniform capping schemes assume that an appropriate number of randomly selected processors on the machine are already capped at $P_{m/c}/N_{max}$, 75W (mid-way between minimum power and TDP), and TDP, such that their aggregate power does not exceed the machine's power budget. The rest of the processors in these configurations are not available to the conventional scheduler in the naïve scheme. PTune+PPartition represents our model that makes variation-aware decisions about scheduling jobs across the entire machine under a machine-level power constraint. The percentages on top of bars indicate how much lower the throughput per naïve scheme is compared to our solution. Our model achieves 5-35% higher throughput.

Figure 3.17 depicts the performance of all the jobs that are scheduled under schemes 1-4 (top left legend) indicated along the x-axis. The y-axis denotes job's performance normalized wrt. to the aggregate job performance (SysIPS) under the respective schemes. We see that our approach (scheme 1) is able to schedule a much larger number of jobs (denser clusters in plot) than the naïve

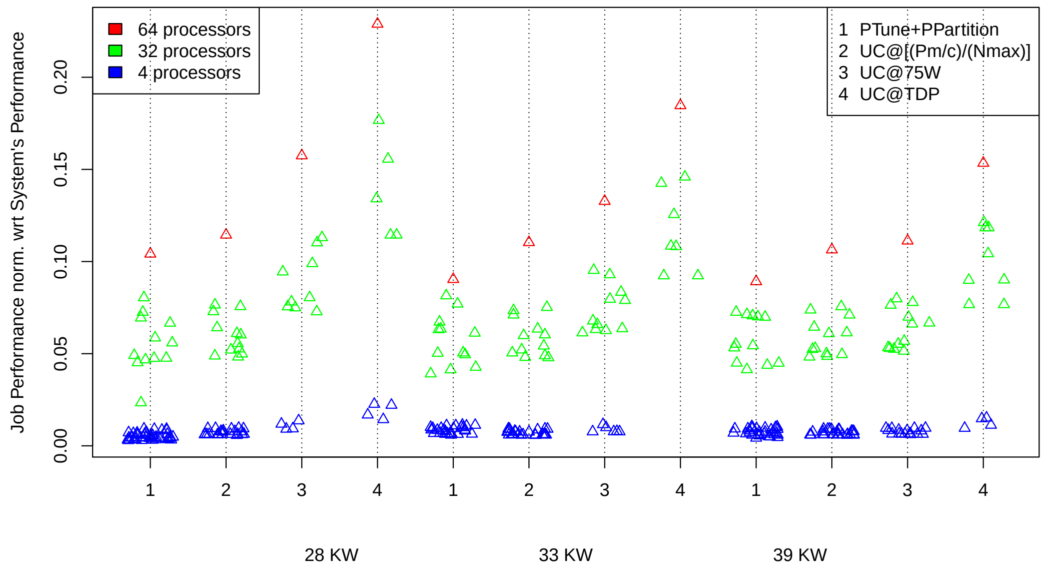


Figure 3.17 Job performance. A job is represented by a triangle

scheduling policy (scheme 2) by trading off performance of some jobs.

3.6 Related Work

Energy has been an important issue in high performance computing (HPC) for over a decade. Supercomputers as old as BlueGene/L have been built with the goal of maximizing power efficiency. Power-scalable clusters that are equipped with voltage and frequency scaling have existed for over a decade that enabled researchers to study the energy problem in HPC. Freeh et al. [Fre05a] investigated the energy-time trade off of MPI applications to prove that it is feasible to save energy by scaling the processor down to lower energy levels with or without time penalty depending on the application. Springer et al. [Spr06] proposed a combined approach of performance modeling and performance prediction for minimizing the execution times of MPI applications under energy bounds. They used voltage and frequency scaling on single cores of a small cluster of up to 10 nodes for their experiments. In addition, there is abundant work presenting algorithms that use frequency and voltage scaling mechanisms for energy savings [Lim06; Rou07; Rou09; Fre05b; Bai15]. In contrast, our work uses power capping via the Intel RAPL interface. Totoni et al. [Tot14; Lan15] presented an ILP-based runtime system that schedules work on a selective *subset* of cores of a single multi-core chip to meet the power or performance constraint. Within-die or core-to-core variation-aware DVFS schemes [**4556740**; **4798265**] have been proposed for chip-multiprocessors.

These schemes select optimal voltage and frequency set points for each of the cores to achieve improved power to performance ratio for the chip. Our work differs from this work in terms of granularity. We study variation across several processors or chips and not across cores of a single multi-core chip. We manage resources at processor (or chip) level. We use either *all or none* of the cores of the chips/multi-core processors on a machine. Our goal is to improve the performance of a parallel job scheduled on multiple processors under a strict power budget.

System-wide solutions for power constraint systems have been proposed that aim at increasing the throughput of systems by leveraging the idea of *hardware overprovisioning* [FF05b; Pat13; Sar13a; Sar14; FF05a]. Sarood et al. [Sar13b] proposed a scheme of determining an optimal number of nodes under strong scaling of applications executing on an overprovisioned system while distributing power between CPU and memory. Etinski et al. [Eti10b; Eti10a] proposed the use of dynamic voltage and frequency scaling (DVFS) at the job scheduling-level to save energy and improve overall job performance. Patki et al. [**patki2**] proposed power-aware backfilling to improve the throughput of the system. Ellsworth et al. [Ell15] presented a power scheduler that enforced a system-wide power bound by reallocating power across the cluster. Our work differs from all of the above because our approach takes the performance variation across processors of a cluster into account while scheduling and tuning jobs for performance.

Inadomi et al. [Ina15] propose performance optimizations across inhomogeneous processors. It provides a detailed analysis of the phenomenon across multiple clusters and provides a set of simple algorithms for intelligent power balancing. These algorithms, while groundbreaking, suffered from two serious limitations. First, the processor power model assumed that CPU clock frequency increased proportionally with power. While that is a useful simplification, our work here shows that the story is not nearly so simple. Second, the algorithms assumed the ideal number of nodes to use was fixed a priori. In our approach, PPart and PTune jointly determine the ideal number of nodes and the power budget for every job at the time of scheduling. While the result of both of our approaches is a power schedule, we are solving a fundamentally different problem.

Kappiah et al. [Kap05] presented a system that saves energy at the expense of execution time by scaling down the frequencies of the cores when they encounter slack time in an MPI application. Rountree et al. [Rou07] used linear programming to establish a bound on optimal energy savings of an MPI application and presented a runtime algorithm to save energy in HPC applications with negligible delay [Rou09]. Power conservation by means of turning off unwanted nodes is proposed in [Pin01]. In the above presented solutions, authors used one core per node and their goal was to maximize energy savings with minimal impact on the execution time. In contrast to these solutions, we are intolerant to performance degradation. We use multicore processors and our goal is to minimize the completion time as long as we stay within the power budget.

3.7 Summary

We presented a hierarchical variation-aware machine-wide solution for managing power on a hardware overprovisioned machine. It consists of a macro-level Power Partitioner that makes power and job scheduling decisions and a micro-level Power Tuner that determines the optimal processor selection and their power caps for a job such that its performance is maximized under a power constraint. PTune achieves up to 29% improvement in performance compared to uniform power capping. It does not lead to any performance degradation, yet frees up to 40% of resources compared to uniform power capping. PPartition is able to improve the throughput of the machine by 5-35% compared to naïve scheduling under the same machine power budget.

We established that under a power constraint, the variability in performance transforms into variation in peak power efficiency. We believe that this variation in power efficiency should be one of the primary considerations in the future power management research.

CHAPTER

4 A POWER-AWARE COST MODEL FOR HPC PROCUREMENT

As we approach exascale, a supercomputer is expected to cost about $200 million and use only 20 megawatts of power in achieving an exaflop [**pcworld**]. Considering the least expensive power in the U.S. ($\approx$ 5 cents/kWh) [**nersc_report**], the cost of operating this machine is $8.76 million per year. Considering a lifetime of five years, its operating cost is about $45 million or nearly 25% of the total cost of ownership (TCO).

4.1 Motivation for power-aware procurement

The status quo of supercomputing procurements is that the power is provisioned for the worst case (WCP). After the initial burn-in phase, which often entailed a Linpack run among other application acceptance tests, the power utilization of the machine drops to 61% of the procured power [Pat15]. This implies that the infrastructure put in place for peak power is no longer fully utilizing this power. Such overprovisioning can be considered a bad investment of budget during steady state operation.

Our prior work on power-efficient computing under manufacturing variations shows that processors are not most power efficient at WCP [**gholkar1**]. Hence, a system with WCP procurement is not power efficient. Processors achieve peak power efficiency at disparate power bounds due to manufacturing variability [Rou12]. Efficient processors achieve peak power efficiency at lower power bounds than the inefficient processors. Increasing the operating power of the processors beyond these bounds leads to diminishing returns in terms of performance. Instead, some power of these processors can be redirected to additional processors to get better performance. Given a hard power constraint, we proposed PTune, a power tuner that exploits this concept to maximize the performance of the system while staying within the power budget.

In this work, we propose a procurement strategy to design a machine that achieves maximum performance per dollar by determining the optimal partitioning of the total budget into capital expenditure (CAPEX) and operating expenditure (OPEX). For the scope of this paper, we limit the OPEX to the cost of power procured for the lifetime of the machine.

Previous research [**Heikkurinen**; **Tak**; **walker**; **Koomey**] has developed cost models for predicting the total cost of ownership (TCO) for cloud computing centers and datacenters. These models do not share our objective of maximizing performance under a fixed system (dollar) budget. In recent work [**TapasyaThesis**], Patki has studied the effect of adding more infrastructure to the system under a fixed power budget. System-wide solutions for power constraint systems have been proposed that aim at increasing the throughput of systems and the runtime of the jobs under a fixed power budget [FF05b; Pat13; Sar13a; Sar14; FF05a; Pat15; Ell15]. Unlike all of this work, we consider the system's power budget as a variable expenditure that we balance against the infrastructure cost to determine the break down of the system's total budget that leads to maximum performance. Furthermore, our model takes the effects of manufacturing variation on the processor's performance into account.

The paper is organized as follows. Section 4.2 states the problem statement. Section 4.3 gives an overview of the cost model. Sections 4.4 describes our procurement strategy. Section 4.5 presents the modeling results. Section 6.6 summarizes the contributions.

4.2 Problem Statement

Given a budget, Sys_Budget, for a system acquisition, design a machine for optimal performance under the assigned budget. The total budget can be divided into two main variable budgets: (1) CAPEX or the cost of the infrastructure ($Sys\_Infrastructure\_Cost$); and (2) OPEX or the cost of power (Sys_Power_Cost) .

We propose a procurement strategy of building a system that achieves maximum performance under the total budget by appropriately partitioning the budget into capital expenditure and operating expenditure. To model the system, we quantify performance in terms of instructions retired per second (IPS). System's performance is represented by $SysIPS$. Our objective is to

$$Maximize(SysIPS) \tag{4.1}$$

subject to $Sys\_Budget \geq Sys\_Power\_Cost +$
$$Sys\_Infrastructure\_Cost.$$

4.3 Cost Model

To demonstrate the model, let us consider a system of Intel Ivy Bridge (12 core Xeon E5-2697 v2 2.7GHz) processors. We make a number of assumptions about the system costs to simplify the problem.

Capital Expenditure (CAPEX):

CAPEX is the cost of the physical assets ($Sys\_Infrastructure\_Cost$). We limit CAPEX to the cost of purchasing racks. The cost of a rack ($Rack\_Infrastructure\_Cost$) is assumed to be \$366K. This number is derived from the data for Jaguar [**jaguar1**; **jaguar2**]. A rack can host a maximum of 100 server nodes, where each node has two processor sockets. In this paper, we assume all racks are identical, i.e., all the racks have the same set of processors and, hence, the same processor characteristics. We assume a variation of 30% in performance across the processors of a rack due to manufacturing variability [**gholkar1**].

Operational Expenditure (OPEX):

OPEX mainly consists of the fraction of the budget spent on power (Sys_Power_Cost) required to run the system for a pre-determined fixed duration. For simplicity, we limit ourselves to the power associated with computing; networking and I/O is subject to future work, and so are secondary operational costs, such as maintenance and support, including staff (typically subject to a separate budget). We assume the minimum power required by the nodes ($P_{node,min}$) is 110W while the maximum power ($P_{node,max}$) corresponding to the Thermal Design Power (TDP) of the processors is 260W. We assume a flat power power of 5 cents per kWh [**nersc_report**]. Finally, we assume the system's lifetime of 5 years ($T = 5 \times 365 \times 24$ hours). This implies that the total cost of acquiring and operating the machine for at least 5 years should not exceed the total budget.

4.3.0.1 Workload:

A supercomputing workload consists of multiple and often coupled parallel scientific simulations that execute on several processors simultaneously. In the interest of simplicity, we assume that the system runs a single application over the duration of its lifetime. To assess the differences between codes, we study the effect of CAPEX and OPEX on performance for 4 different codes, viz., EP, SP, BT from the NAS parallel benchmark suite and CoMD from the Mantevo suite, in the modeling results.

4.4 Procurement Strategy

We propose a procurement strategy that builds a system with maximum performance under an assigned total budget. We assume that a rack is always filled to capacity with compute nodes, i.e., it hosts 200 processors. It can be powered at:

- Maximum Power: A rack is supplied with worst case power (WCP) to be able to run the processors at their Thermal Design Power (TDP).
- Minimum Power: A rack is supplied with minimum power required by the computation capability to be functional.
- Medium Power: A rack is supplied with less than required maximum power but greater than the bare minimum power required by the computation capability it hosts.

The performance (and cost) of a rack is maximal under maximum power configuration. The maximum power that can be provisioned to a rack ($Rack\_max\_power$) is $100 \times P_{(node,max)}$. Its power cost is $100 \times P_{(node,max)} \times \$0.05 per kWh \times T = \$56,940$. Therefore, $Rack\_TotalCost_{(max)} = \$56,940 + \$366,197 = \$423,137$.

$Rack\_TotalCost_{(min)}$ represents the minimum budget required to add a rack to the system. The infrastructure cost is the same as that of a packed rack. Its power cost is $100 \times P_{(node,min)} \times \$0.05 per kWh \times T = \$21,900$. Therefore, $Rack\_TotalCost_{(min)} = \$22,338 + \$366,197 = \$388,535$.

Given a fixed Sys_Budget, the number of racks in the system (num_rack) that can be procured ranges between $num\_rack_{\perp}$ and $num\_rack_{\top}$.

The lower bound on num_rack ($num\_rack_{\perp}$) is
$num\_rack_{\perp} = \frac{Sys\_Budget}{Rack\_TotalCost_{(max)}}$.

The upper bound on num_rack ($num\_rack_{\top}$) is
$num\_rack_{\top} = \frac{Sys\_Budget}{Rack\_TotalCost_{(min)}}$.

For num_rack racks, the $Sys\_Infrastructure\_Cost$ is
$num\_racks \times Rack\_Infrastructure\_Cost$.

The Sys_Power_Cost is calculated as

$$\begin{aligned} Sys\_Power\_Cost = Sys\_Budget \\ - Sys\_Infrastructure\_Cost. \end{aligned}$$

The Sys_Power is calculated as

$$\begin{aligned} Sys\_Power = min(\frac{Sys\_Power\_Cost}{\$0.05\ per\ kWh \times T}, \\ num\_rack \times Rack\_max\_power). \end{aligned}$$

The $Rack\_Power$ is calculated as
$Rack\_Power = \frac{Sys\_Power}{num\_rack}$.

At rack-level, for an assigned power budget of $Rack\_Power$, PTune[**gholkar1**], our variation-aware power tuner, determines the power distribution across the processors that achieves maximum performance within this rack.

$Rack\_Performance = PTune(Rack\_Power)$

As all racks are considered to be identical (statistically, given their large processor count), the system's performance can be calculated as

$SysIPS = num\_rack \times Rack\_Performance.$

Given a $Rack\_Power$ budget, PTune[**gholkar1**] distributes the budget across the processors of the rack systematically in a variation-aware, i.e., processor-sensitive, manner to maximize the performance of this rack. A system configuration can be represented as a tuple over (1) the number of racks in the system, (2) power allocated to a rack, and (3) the resulting performance of a system, denoted as ($< num\_racks, Rack\_power, SysIPS >$). The winning design is the one that achieves the objective of Eq.4.1.

4.5 Experimental Setup

Characterization experiments were conducted on the *Catalyst* cluster, a 324-node Ivy Bridge cluster at Lawrence Livermore National Laboratory (LLNL). We used the performance and power data from this entire cluster to represent one rack in the machine that we designed in the previous section. Each node has two 12-core Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz processors and 128 GB of memory. We used MVAPICH2 version 1.7. The codes were compiled with the Intel compiler version 12.1. The msr-safe kernel module provides direct access to Intel RAPL registers via libmsr [Sho14]. We used the package (PKG) domain of RAPL that provided us the capability of capping power for each of the processors in an experiment. The environment was simulated in R. We used EP, BT, and SP from the NPB suite and CoMD from the Mantevo suite in their pure MPI versions.

4.6 Results

We observe the impact of different breakdowns of the system's budget (into CAPEX and OPEX) on the performance of the system. At the rack-level, we use PTune [**gholkar1**] to get the maximum performance under the rack's power budget. Figure 4.1 depicts the performance of a rack at several power budgets. The x-axis represents the rack's power budget in watt and the y-axis represents the rack performance normalized to the maximum performance across applications. We show results for EP, SP, BT, and Comd. Data points corresponding to 26kW represent the performance at maximum power ($Rack\_max\_power$). Every data point represents the performance corresponding to variation-aware power tuning of the rack power across processors. We observe that the performance increases non-linearly with rack power.

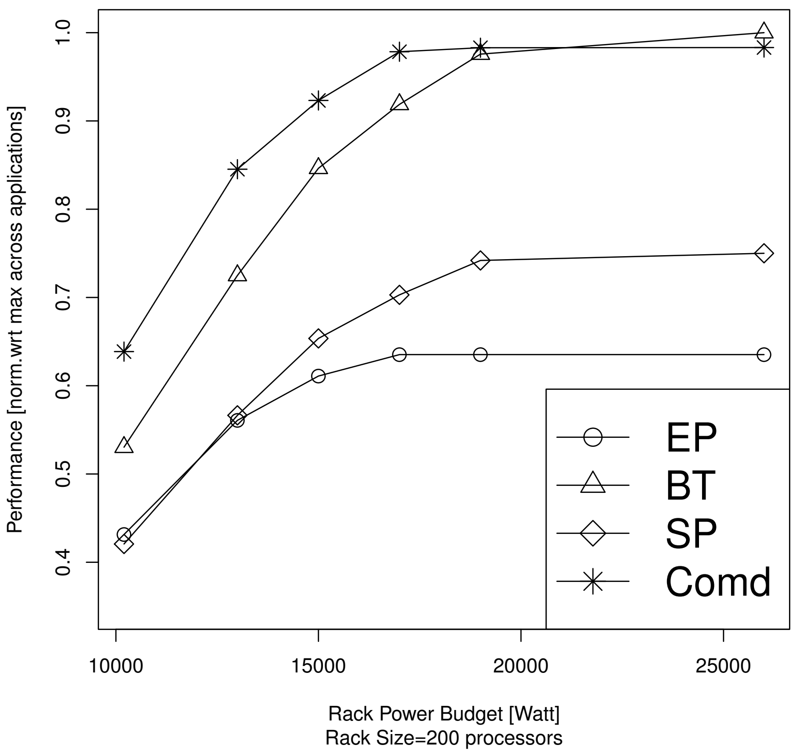


Figure 4.1 PTune: Power Tuning Results for a rack at several rack power budgets.

Figure 4.2 shows the performance of a system under a fixed budget of \$102 million. The x-axis represents the number of racks and the y-axis represents the system performance normalized to the optimal performance. The optimal design at 252 racks achieves 5% better performance compared to the WCP. Purchasing more racks beyond this point (and effectively procuring less power) leads to suboptimal capability. However, increasing the number of racks up to 255 gives at least as much performance as the WCP design but addition capacity.

Purchasing more than 255 racks degrades the capability of the machine by up to 29% compared to the optimal design. No more racks can be added beyond this point as the system budget will not suffice to provision the bare minimum power required to power the system. It is important to note that rack $(n+1)$ is procured at the expense of the power stolen from the prior n racks. Since the racks are always fully packed, their CAPEX is fixed. Hence, the only way to accommodate an addition rack is to reduce the OPEX of the prior n racks. From this, we conclude that aggressively purchasing infrastructure under a fixed system budget by disregarding the diminishing budget for power does *not* lead to the best capability system design. A *balance* needs to be stricken between

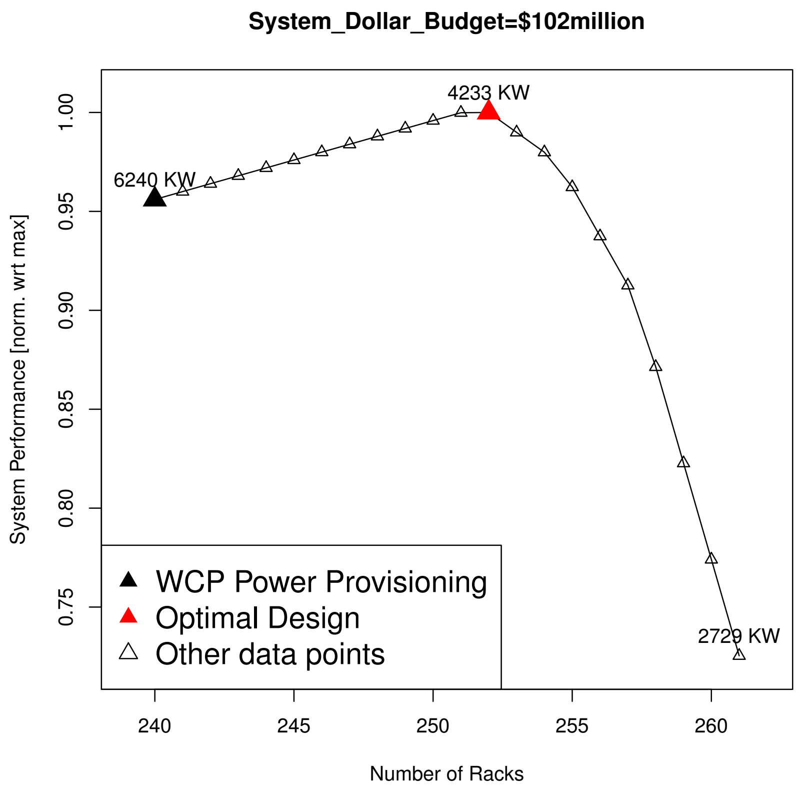


Figure 4.2 Effect of budget partitioning on the overall system performance

the CAPEX and the OPEX for an optimal system design.

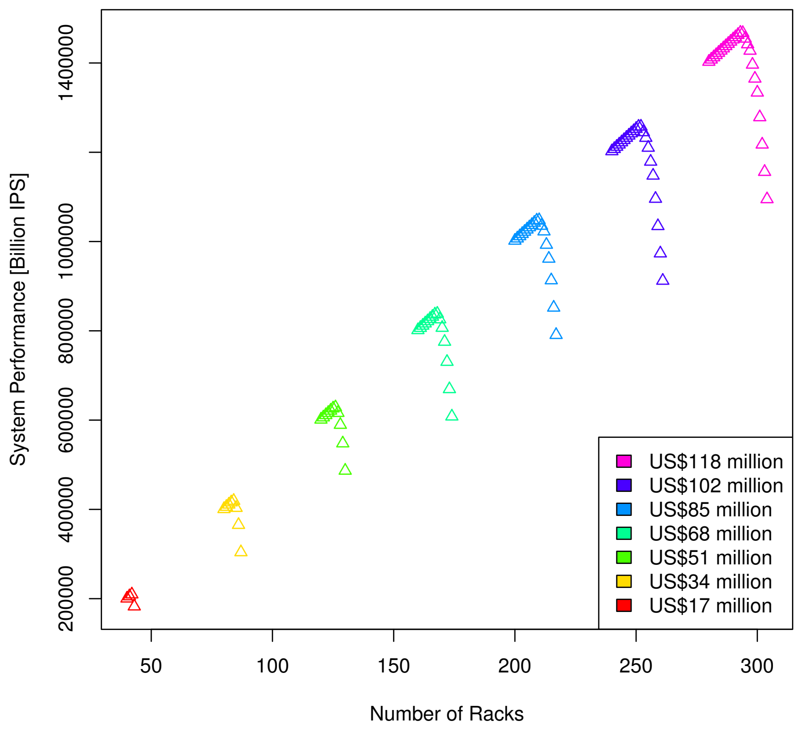


Figure 4.3 EP

Figures 4.3, 4.4, and 4.5, compare the results for 3 different codes under several system budgets. (Results for SP are similar to EP and are omitted due to space). The x-axis represents number of racks and the y-axis represents system performance. The figure shows 7 system budgets that correspond to the cost of 40, 80, 120, 160, 200, 240, 280, and 320 racks at maximum power. The first data point in each system budget curve represents the performance at WCP.

Overall, we make the following observations for these figures:

• Power-aware procurement can improve performance by up to 4%, 6%, 7%, and 4% for EP, SP, BT, and Comd, respectively, compared to worst-case power provisioning.

• Without power-aware procurement, performance can degrade up to 25%, 28%, 41%, and 38% for EP, SP, BT, and Comd, respectively, compared to worst-case power provisioning.

• Performance increases linearly with the system budget.

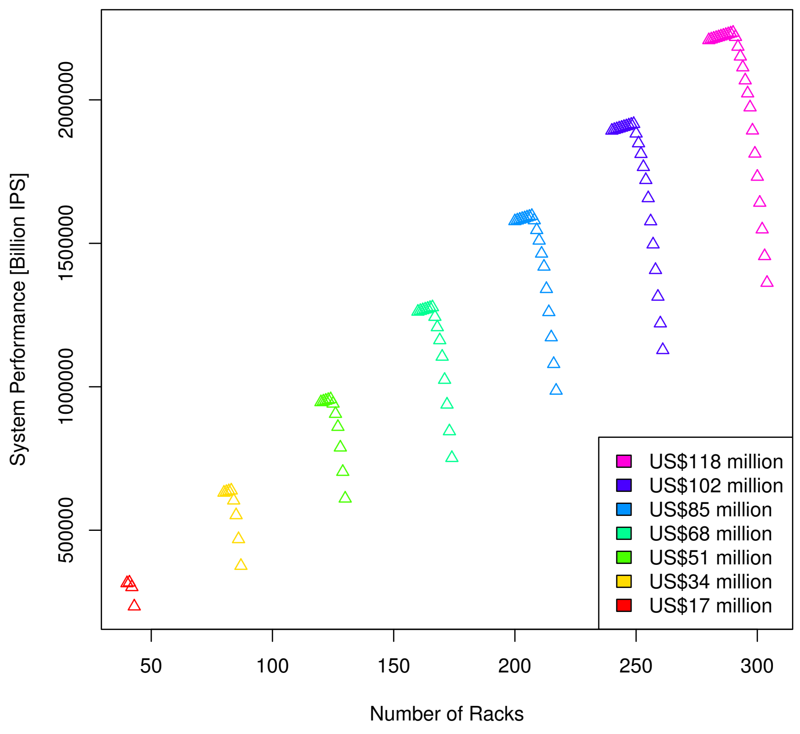


Figure 4.4 BT

• Performance linearly increases with the number of racks before it reaches the peak, after which there is a steep performance degradation. Peak performance is achieved relatively early in case of BT (Figure 4.4) compared to EP and CoMD (Figures 4.3 and 4.5). BT has a longer tail that follows the peak, representing the feasible system designs that lead to degraded system performance due to aggressive infrastructure procurement.

• The absolute performance achieved by the system depends upon the application, i.e., the optimal system design also depends on the application. The assumption that our system executes a single application during the course of its lifetime does not hold in reality. Hence, it is necessary to compromise on middle ground with a design that, on *average*, fits the needs of the applications when run in capability mode.

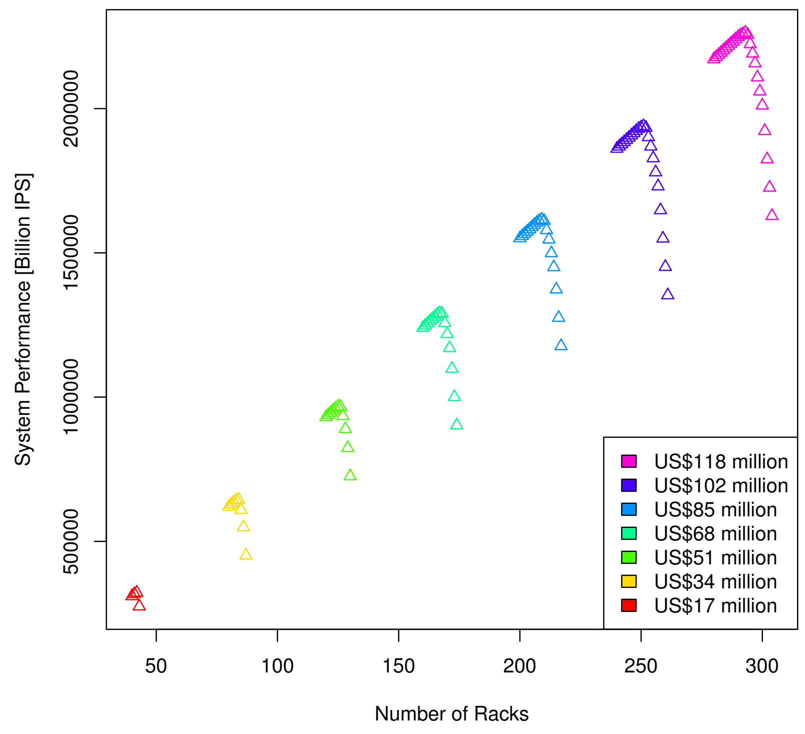


Figure 4.5 Comd

4.7 Summary

This work analyzed the effect of manufacturing variations on procurement and operations. It showed that when partitioning the system's budget into infrastructure cost and power cost, a balance needs to be stricken between the two to achieve an optimal ratio of performance per cost (dollar). More infrastructure does not necessarily mean more performance under a fixed total budget. Model-based analysis provides the means to optimize power-aware procurement/operation for a set of application codes. Such strategies may need to be adopted in the future to best utilize a compound budget for systems and operations. Failure to plan ahead may either result in a nominal loss in performance compared to such a balanced system, or a significant additional cost will be incurred in operating cost to efficiently utilize an overprovisioned hardware installation.

CHAPTER

5

POWER SHIFTER: A RUNTIME SYSTEM FOR RE-BALANCING PARALLEL JOBS

The Uniform Power scheme (described in Chapter 2) is a naïve approach of enforcing a job's power budget. It suffers from two major drawbacks. First, processors on a cluster exhibit variations in power efficiency that translate to variations in performance [Rou12; Gho16], which makes the cluster non-homogeneous. Second, in practice, parallel applications tend to experience computational imbalance across their processes (worker threads) due to uneven workload distributions. As the simulation progresses, workloads can shift between processes leading to dynamically load imbalance. Due to load imbalance, some processes finish computation early and wait for others at barriers, collectives or blocking calls even on homogeneous systems. A non-homogeneous system under a uniform power scheme can actually *worsen* performance if an overloaded process is mapped to an inefficient socket, which lengthens its computation phases. This leads to longer barrier wait times for underloaded processes. Such waiting results in unused CPU cycles at barriers and other collectives, which further contributes to energy wastage. This is depicted in Figure 5.1.

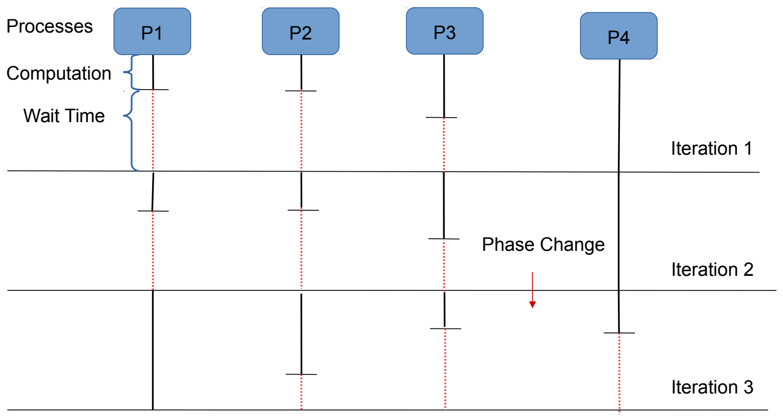


Figure 5.1 Computational imbalance in a parallel job

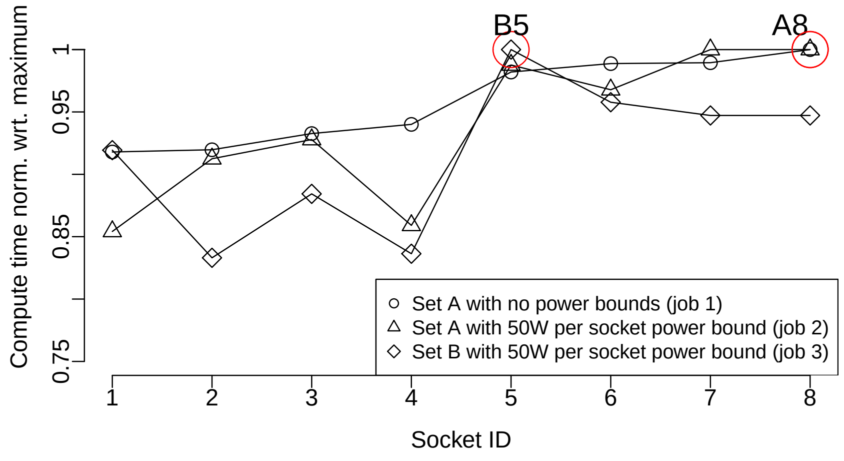


Figure 5.2 Unbounded compute times predict bounded times poorly.

5.1 Motivation for a power management runtime system

Prior work, PTune has addressed the problem of maximizing job's performance under a power budget. PTune is a processor variation-aware *static* job power management solution but it does not handle the load imbalance. The application's computation time profile differs across disjoint sets of sockets under a power bound making it difficult to derive a static solution. This is depicted in Fig. 5.2. The x-axis represents socket identifiers and the y-axis represent computation time between two synchronization calls normalized with respect to the maximum compute time across all sockets of a job (lower is faster). The application runs on 8 sockets. The socket finishing last, i.e., the socket on the critical path, is circled.

The graph shows three jobs running on two distinct sets (A and B) of sockets. The job on Set A with no power bounds is represented by circles and A8 is the performance bottleneck. The job with an average socket power bound of 50W on set A is represented by triangles and A8 is its performance bottleneck. The job on Set B with an average socket power bound of 50W is represented by diamonds and B5 is its performance bottleneck. The mapping of processes to socket IDs is identical in all three jobs.

We observe the following:

• Uneven work distribution in a parallel job leads to different compute times across sockets leading to wait times and wasted cycles and energy.

• The compute time curve with no power bounds (uniform performance) is different from that with power bounds on sockets. For example, under power bounds, socket 4 finishes before sockets 2 and 3 unlike the no power bounds case.

• The compute time curve with identical power bounds on two different sets of sockets, A and B, is different. The performance bottleneck shifts from socket A8 to socket B5. This is because B5 (set B) is more power efficient than socket A8 (set A). Hence, under identical power bounds, the former completes the same amount of computation significantly faster than the latter.

We address the inadequacies of PTune and other existing power [Mar15] assignment solutions with Power Shifter (PShifter), *a runtime system* that maximizes job's performance without exceeding its assigned power budget. Determining a job's power budget is beyond the scope of PShifter. PShifter takes job power budget as an input.

PShifter is a hierarchical closed-loop feedback controller that makes measurement-based power decisions at runtime and adaptively. At the top level, the cluster agent monitors the power consumption of the entire job. It opportunistically improves the job's performance by feeding its unused power budget back into the job. Each socket is periodically monitored and tuned by a local agent. A local agent is a proportional-integral (PI) feedback controller that strives to reduce the energy wastage by its socket. It gives up socket power when its socket is an "early bird" that waits at blocking calls and collectives. The cluster agent senses this power dissipation within a job in its monitoring cycle and effectively redirects the dissipated power to where it can best improve the overall performance of the job (i.e., reduce the critical path). In experiments, PShifter achieves a performance improvement of up to 21% and energy savings of up to 23% compared to the naïve approach.

Unlike PTune [Gho16] that relied on prior characterization data and was agnostic of phase changes in computation, PShifter makes decisions based on runtime measurements and it transparently and automatically applies power capping non-uniformly across nodes of a job in a dynamic manner adapting to phase changes. It can readily be deployed on any HPC system with power capping capability without modifications to the application's source code.

5.2 Design

PShifter is a hierarchical, closed-loop feedback controller that operates at runtime alongside the job. The key idea is to make measurement-driven decisions dynamically about moving power within a job from where it is wasted to where it is required (on the performance-critical path).

5.2.1 Closed-loop feedback controller

The general idea of a closed-loop feedback controller is depicted in Fig. 5.3. *System* is the component whose characteristics are to be monitored and actuated by the controller. The current state of the system is defined in terms of the *process variable* (PV). The desired state of the system, PV=K, where K is a constant, is called the setpoint (SP). It is input to the feedback controller. A feedback-control consists of three main modules, viz., sensor, control signal calculator, and actuator.

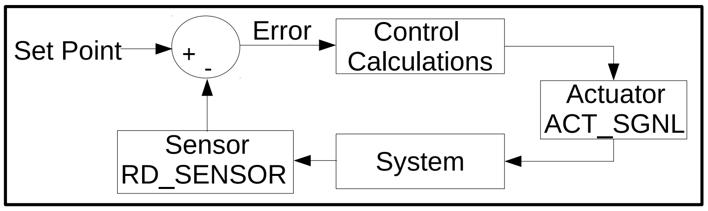


Figure 5.3 Closed-loop Feedback Controller

Sensor: The sensor periodically monitors the state of the system defined in terms of process variable. This is indicated by RD_SENSOR.
Control Signal Calculator: The error (err) in the system state is calculated as the difference between the setpoint and the measured process variable. The feedback controller calculates the feedback to the actuator as a function of this error. It also takes into account the history of error values. The output of the feedback-control loop is calculated as the sum of the previous values of the actuator and the calculated feedback.
Actuator: The actuator applies the calculated signal to the system. This is indicated by ACT_SGNL.

5.2.2 PShifter

PShifter consists of a dual-level hierarchy of closed-loop feedback controllers, a local agent and a cluster agent. Fig. 5.4 depicts the high-level architecture of the controller and the mapping of its components to the components of the underlying system.

Fig. 5.4(a) shows the architecture of a typical HPC cluster. It consists of multiple server nodes.

Each server node hosts one or more sockets. Our server nodes have two sockets, each hosting a single 12-core processor. Multiple parallel jobs spanning across one or more nodes can run on a cluster, e.g., job1 runs on the top 4 nodes while job2 runs on bottom 4 nodes. An instance of PShifter runs alongside each job and enforces its power budget while improving job performance under the power constraint.

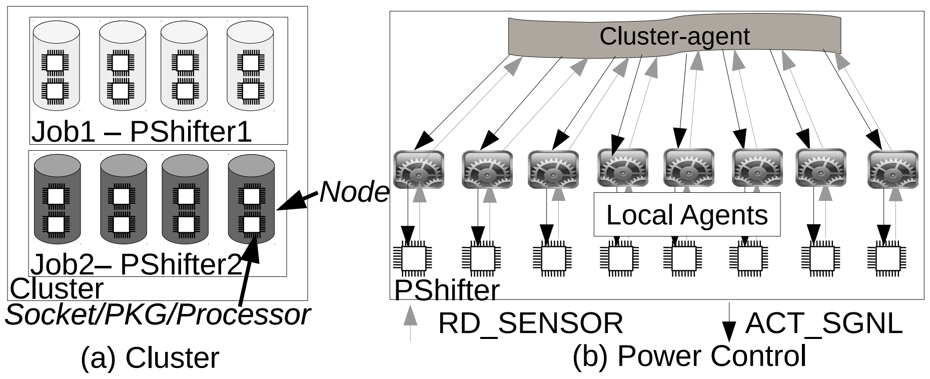


Figure 5.4 PShifter Overview

PShifter's controller hierarchy is depicted in Fig. 5.4(b). It consists of a cluster agent at the root and several local agents at the leaves. At the bottom level, several local agents monitor the performance and manage the energy consumption of the individual sockets of the job. At the top level, the cluster agent overlooks the power consumption of the entire job.

Initialization

At job initialization, the cluster agent enforces a job's power budget by uniformly distributing the power across its sockets. This is the system state established by the naïve scheme. It enforces the job level power budget but it does not address the problem of performance degradation due to imbalanced jobs. Starting form this initial state, PShifter (local and cluster agents) gradually moves power within the job to change the unbalanced performance state of the system to a desired, more balanced performance.

Local agent

Local agents are the leaf nodes in PShifter's hierarchy and implement a closed-loop feedback controller. A local agent monitors and controls performance and energy consumption of a unique socket in the job allocation. There is one local agent per socket in the job allocation. The local agents are invoked periodically and asynchronously. The process variable, PV, for the local agent is defined as the ratio of the computation time to the total time (including computation, wait, and communication time) between two subsequent invocations.

$PV = \frac{ComputationTime}{TotalTime}$

It is a measure of the socket's computational load. In other words, it measures the proportion of the total time spent doing useful work. Sockets with comparatively lower PV are called *underloaded sockets* while sockets with higher PV re called *overloaded sockets*. Notice that computation time includes memory access time. Thus, the process variable captures the memory-boundedness of the job.

A local agent uses the power cap of the associated socket as its actuator. Each socket hosts multiple (12) processes, each pinned to a unique core. Pinning processes to cores avoids unnecessary overhead of Linux process migrations and longer memory access delays due to remote Non-Uniform Memory Accesses (NUMA). The PV_p is measured by each process and for each socket's local agent. The process with the maximum PV_p is called the representative process of the socket and the socket's PV is equal to the representative process's PV_p. The processes that have shorter computation tasks finish early and wait for other processes (with longer computation tasks) before engaging in communication or synchronization. These wait times lead to wastage of power that is not utilized for any work (computation). The PV of a socket with "early bird" processes tends to be much less than that of sockets with a bottleneck process, which tend to have a PV of almost one (as communication takes some finite time). The goal of the local agent is to maximize the PV (Note: $PV \leq 1$) and thus to minimize the wait time. The setpoint (SP) is initialized to 0.95. It is experimentally determined. The local agent strives to achieve this setpoint by lowering the power of the socket with PV < 0.95. This slows down the processes that originally had shorter computation tasks and subsequently reduces their wait times in future iterations. It is important to note that local agents make decisions based only on local information, i.e., they do not need to communicate or synchronize with others.

The local agent is implemented as a proportional-integral (PI) controller. The PI controller makes power decisions dynamically at runtime based on measured values of the process variable. The PI feedback from the local agent is calculated as

$fb_{local\_agent} = Pterm + Iterm,$

where Pterm denotes the proportional term and Iterm stands for integral term. Pterm is calculated as

$Pterm = Kp * e(t),$

where Kp is a constant. Iterm is calculated as

$Iterm = Ki * \int e(t)dt,$

where Ki is a constant. The error, (e(t)) at time t, is calculated as

$err = PV - SP.$

The constants were application specific (but vary only 5% between applications) and were determined experimentally, a common method in feedback-control systems: We set the integral gain, Ki, to zero and increase the proportional gain, Kp, until the outputs oscillate. We then increase Ki to reduce the steady state error to an acceptable level of 5%.

Pterm accounts for the current error and calculates a response proportional to this error. Iterm is calculated as the product of aggregated past errors over time and the constant Ki. Iterm is dependent on the magnitude of past errors and the time for which they stay uncorrected. If the output generated by the proportional term is small and does not reach the setpoint over multiple invocations, the error aggregated over several invocations helps in strengthening the output (larger Iterm) and, in turn, approaches the setpoint faster. This may cause the controller output to overshoot the setpoint. Hence, we use a bandpass filter to limit Iterm. Error values are accumulated for Iterm only until it reaches the upper or lower (error can be positive or negative) saturation point of the filter. The power cap of a socket is the actuator. The output of the local agent or the new power cap is calculated as

$Pcap_{(current)} = Pcap_{prev} + fb_{local\_agent}$,

where $Pcap_{prev}$ is the previous power cap of the socket and fb_{local_agent} is the calculated feedback.

The impact of power modulation on the socket's performance or PV depends on the memory-boundedness and the CPU-boundedness of a job. As the local agent monitors PV and keeps track of the error history as it sets power caps at each invocation, it indirectly learns about the nature of the job. It incorporates this knowledge into feedback in the form of Iterm.

Cluster agent

The PV of the cluster agent is the job's power consumption measured using Intel's RAPL interface, i.e. the aggregate power consumption of all of its sockets. The cluster agent's SP is set to the job's power budget (P_{job_budget}). where P_i is the measured power consumption of the i^{th} socket. Note that the error is indicative of a job's unused power (P_{job_unused}) budget. The cluster agent distributes this unused power uniformly across all sockets. The feedback from the cluster agent ($fb_{cluster\_agent}$) is calculated simply as

$fb_{cluster\_agent} = \frac{P_{job\_unused}}{N}$,

where N is the total number of sockets of the job.

The cluster agent and the local agents share their actuators, i.e., the cluster agent also uses the power actuators of the sockets. When the cluster agent is invoked, it overwrites the power caps of sockets. The new power caps are calculated as $Pcap_{current} = Pcap_{prev} + fb_{cluster\_agent}$. The cluster agent is invoked less frequently than the local agent giving the local agent multiple opportunities to give up just enough power to reach the local setpoint. When the cluster agent is invoked, it effectively feeds the unused power back into the job by redistributing it uniformly across sockets. This step results in redirecting some power from the sockets that have shorter computation to the sockets that have longer computation (bottleneck processes that determine the completion time of the job) in every invocation. This opportunistically speeds up bottleneck processes and reduces the overall completion time of the job.

Fig. 5.5 summarizes the design of PShifter. The local agents of sockets 2, 3, and 4 reduce the power of the sockets to slow down the processing resulting in reduced wait times. When the cluster agent is invoked, it calculates the unused power of the job and distributes this uniformly across all of its sockets. As a result, socket 1, which never gave up any power, now gets some additional power to finish its computation faster, resulting in an overall improvement in performance.

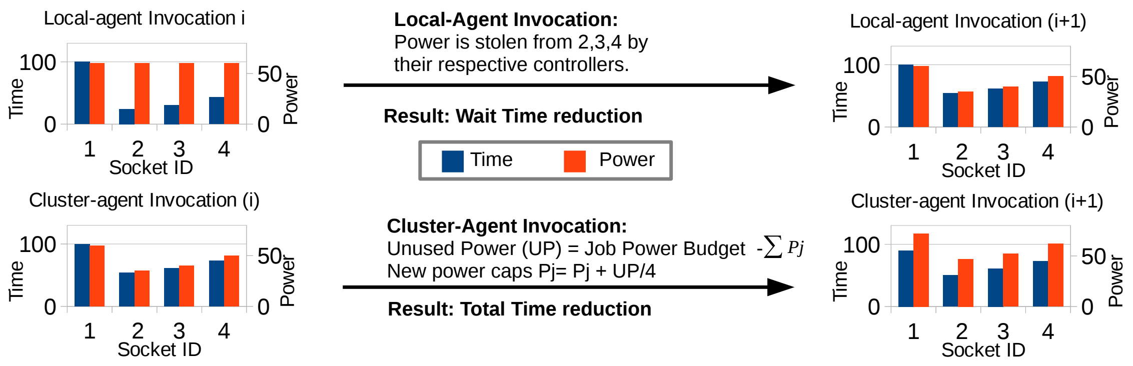

Local-agent Invocation i
Local-Agent Invocation:
Power is stolen from 2,3,4 by their respective controllers.
Result: Wait Time reduction
Local-agent Invocation (i+1)
Time
Power
Socket ID
Cluster-agent Invocation (i)
Cluster-Agent Invocation:
Unused Power (UP) = Job Power Budget -∑ Pj
New power caps Pj= Pj + UP/4
Result: Total Time reduction
Cluster-agent Invocation (i+1)

Figure 5.5 Cluster agent and Local agent

5.3 Implementation and Experimental Framework

Our solution was developed on the *Catalyst* cluster at Lawrence Livermore National Laboratory (LLNL). It is a 324-node Intel Ivy Bridge cluster. Each node has two 12-core Intel(R) Xeon(R) E5-2695 v2 @ 2.40GHz processors and 128 GB of memory. Each MPI job runs on a dedicated set of nodes on this cluster. No two jobs share nodes. For the power measurement and actuation, we leverage Intel's Running Average Power Limiting (RAPL) feature [Int11]. From Sandy Bridge processors onward, Intel supports this interface that allows the programmer to measure and constrain the power consumption of the package (PKG) by writing a power limit into the RAPL model specific register (MSR). Here, a package is a single multi-core processor chip or a socket. RAPL is implemented in hardware. It guarantees that the power consumption stays at the power limit specified by the user. The msr-safe kernel module installed on this cluster enabled us to read from and to write to the Intel RAPL model specific registers in userspace via the libmsr library [Sho14].

We used MVAPICH2 version 1.7. The codes were compiled with the Intel compiler version 12.1. PShifter is a library that can be linked with the application. It is implemented using the MPI standard profiling interface[MG97] (PMPI). The feedback controllers are called in the wrapper functions of MPI calls, which are invoked by an MPI application. Hence, our solution does not require any modifications to the application. Applications only need to be linked to our library. We pin each MPI process to different cores of the job's sockets. The term process is used to refer to an MPI process.

At initialization, i.e., in the MPI_Init wrapper, PShifter sets the RAPL power cap of all the sockets within a job to $\frac{Job's PowerBudget}{N}$. It creates two types of MPI sub-communicators, viz., MPI_LOCAL_COMM per socket and MPI_CLUSTER_COMM per job using MPI_Comm_split. Each MPI_LOCAL_COMM communicator consists of all the processes pinned to the cores on the same socket. The process pinned to core 0 acts as sub-root (sub-rank 0) and runs the local agent. The MPI_CLUSTER_COMM consists of all the local agents with the local agent at MPI rank 0 acting as the root or the cluster agent. Each process records its computation time and the total time (computation+wait time). It then calculates its PV at every invocation. The local agent of every socket selects the process with maximum PV across all its processes as a representative. It also measures the power consumption of its socket. The local agent computes fb_{loc} using this PV and sets the new power cap for its processor. To set the power cap, it calls set_rapl_limit of the libmsr library.

The cluster agent invocation involves three steps. First, it measures the power consumption of the job by aggregating the power consumption across its sockets. This is done using MPI_Reduce over the MPI_CLUSTER_COMM communicator. The feedback is calculated and broadcasted (using MPI_Bcast) by the cluster agent to all the local agents that enforce the new power caps.

5.4 Experimental Evaluation

We evaluated PShifter with MiniFE and CoMD proxy applications from the Mantevo [Her09] benchmark suite, and a production application, ParaDiS [Bul04]. miniFE is representative for unstructured finite element codes. CoMD is a proxy application for molecular dynamics codes. Explicit load imbalancing was turned on for these proxy applications. ParaDiS [Bul04] is a dislocation dynamics simulation code. We weakly scaled the inputs to miniFE and CoMD by increasing the input sizes proportionally to the number of nodes. For ParaDiS we used two LLNL inputs, small scale ($\leq$ 384 cores) and large scale ($>$ 384 cores). We weakly scaled each of these two inputs as we ran them on up to 32 and 256 sockets, respectively.

We used the MPI versions of these codes in our experiments. We report performance in terms of job completion time in seconds, average power in Watts, and energy in Joules. The reported numbers are averages across five runs. with a maximum standard deviation of 7%. The baseline for evaluation is uniform power (UP), where the job's power budget is distributed uniformly across all the sockets of a job. The maximum socket power consumption across our applications was observed to be 90, i.e., none of these benchmarks, would ever exceed 90W on a socket. Hence, we show evaluation results for power constraints ranging from 90% to 60% (80W to 55W, respectively) of maximum power.

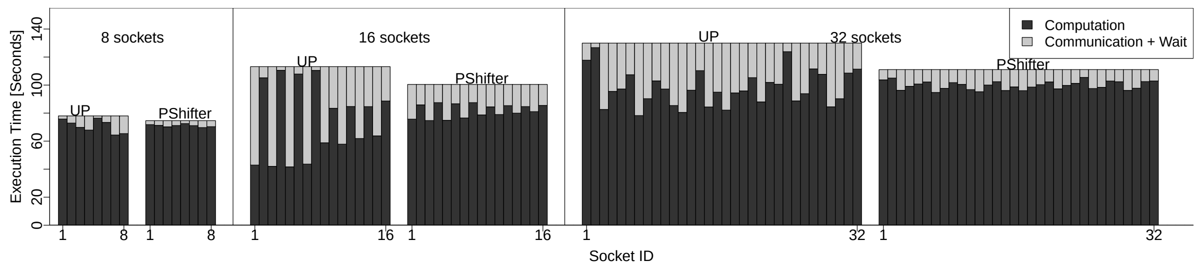


Figure 5.6 Performance Imbalance across a job of miniFE. The job's power budget is set as 55W × #Sockets

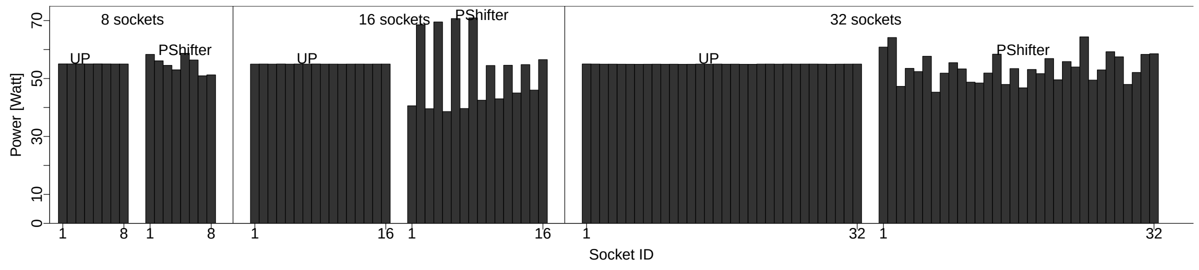


Figure 5.7 Average power consumption by different sockets of a job for miniFE. The job's power budget is set as 55W × #Sockets

5.4.1 Comparison with Uniform Power (UP)

Fig. 5.6 shows the performance imbalance that persists within 8, 16, and 32 socket jobs of miniFE. The job's power budget is set to 440W, 880W, and 1760W for 8, 16, and 32 socket jobs, respectively. The y-axis represents execution time of the job in seconds. Each stacked bar in the plot represents the computation (gray) and non-computation (white) time for each socket's representative process. The total height of the bars (gray+white) indicates the completion time of the job. For each job size, there are two groups of bars, one corresponding to the UP scheme and the other corresponding to the PShifter scheme as labeled in the plots. The x-axis represents the socket IDs within each job. It can be observed that some sockets spend more time in the computation phase than other sockets causing performance imbalance within the job. This imbalance can be quantified as $I = \frac{max(C_i)-min(C_i)}{mean(C_i)}$, where C_i is the computation time of the i^{th} socket's representative process.

Observation 1: PShifter moves power from underloaded sockets to overloaded sockets.

For example, in case of the 16 socket job, it can be observed that under UP, even numbered sockets have longer computation phases and, hence, high PVs (overloaded sockets) than odd numbered sockets (underloaded sockets). PShifter effectively moves power from underloaded sockets to overloaded sockets within the same job.

This is shown in Fig. 5.7. The y-axis represents average power consumption in Watts over the same x-axis as before (socket IDs). UP indicates uniform power distribution across different sockets of a job while PShifter enforces a non-uniform distribution of the job's power budget, with underloaded sockets (e.g., odd sockets in the 16 socket job) at lower power than overloaded sockets (e.g., even sockets).

Observation 2: PShifter leads to a balanced execution of a parallel job.

As a result of shifting power from underloaded sockets to overloaded sockets, the rate of computation on the underloaded sockets slows down while that on the overloaded sockets speeds up. This leads to a reduction in the computation time on the overloaded sockets and an increase in the computation time on the underloaded sockets, resulting into a balanced parallel execution. The wait times for underloaded sockets are also reduced under PShifter. This can be observed in Fig. 5.6 by comparing the corresponding UP and PShifter bar plots for 8, 16 and 32 sockets.

Table 5.1 summarizes the imbalance values for each of the jobs. PShifter reduces the imbalance by 75%, 87%, and 80% for 8, 16, and 32 socket, respectively.

Table 5.1 Imbalance Reduction

Socket Count	Imbalance with UP	Imbalance with PShifter
8	16%	4%
16	31%	4%
32	49%	10%

Observation 3: PShifter reduces the completion time of the power-constrained parallel job without violating its power budget.

The completion time of a parallel job is constrained by the completion time of the socket with the largest PV value as it hosts the bottleneck process with the longest computation time. Such a socket is on the critical path of parallel execution. Moving power to this overloaded socket speeds up its rate of computation and thus reduces the length of the critical path of parallel execution and the job's completion time. This can be observed in Fig. 5.6, where PShifter reduces the completion times by 2%, 10%, and 14%, for 8, 16, and 32 socket jobs, respectively.

Observation 4: The power consumption of sockets under PShifter is proportional to the respective computational loads (or computation times under UP).

It is interesting to note that the shape of the computation time curve under UP in Fig. 5.7 matches the power curve under PShifter for each job. This shows that PShifter shifts just the right amount of power between sockets such that the resulting power consumption of each socket is proportional to its computational load. However, it is important to note that PShifter does not need any prior information about the job's computation time profile. Instead, the decisions about the amount of power to be shifted, the source and the destination of shifted power are made by the local agents and the cluster agent together based on their runtime sensor inputs.

Observation 5: PShifter reduces the energy consumption of the parallel job.

PShifter reduces the energy consumption of 8, 16 and 32 socket jobs by 2%, 14%, and 16%, respectively, over UP. This has two aspects. First, PShifter reduces the completion time of the job without exceeding its power budget. This leads to energy savings. Appropriate shifting of power by PShifter leads to a reduction in wait times of the underloaded sockets. Also the power assigned to the underloaded sockets is lower under PShifter. Hence, while they wait for reduced durations, they consume lower energy even in their waiting phases compared to that in case of UP. This further adds to the energy savings.

Scalability

We evaluated PShifter on up to 3072 cores on 256 sockets. Fig. 5.8 - 5.10 compare the performance of miniFE, CoMD, and ParaDiS under UP and PShifter at six different job power budgets for each job size. The x-axis denotes the number of sockets (n) in a job. Average power per socket (Avg_Pow) is indicated at the top of the plot. A job's power budget is set at $n \times Avg\_Pow$. The y-axis represents a job's completion time in seconds. As indicated in the legend, the performance under UP and PShifter is represented by gray and white bars, respectively. The percentages on the top of the PShifter bars indicate performance improvement or reduction in completion time achieved by PShifter over UP. The error bars show the maximum and minimum completion times across five repetitions of each experiment.

PShifter achieves performance improvements of up to 17%, 21%, and 21% for miniFE, CoMD, and ParaDiS, respectively. The geometric mean is 6.5%, 5%, and 7.5% for Fig 5.8 - 5.10, respectively. The overhead of running PShifter is no more than 5% (included in the results). In most cases, PShifter causes less run-to-run variation across several repetitions of every experiment compared to UP. This can be observed by comparing the error bars on PShifter and UP in Fig. 5.8 - 5.10.

In some exceptional cases, we observe 0% performance gains. For example, consider the 128 socket experiments of miniFE with an average socket power of 55W. PShifter reduces the imbalance by 32% over UP. However, this is countered by the added communication time leading to 0% improvement in performance. Nonetheless, PShifter never loses in performance.

Fig. 5.11 - 5.13 compare the energy consumption of miniFE, CoMD, and ParaDiS under UP and PShifter for the six different job power budgets as before. The x-axis is the same as in case of Fig. 5.8 - 5.10. The y-axis represents the energy consumption in KiloJoule. The percentages on top of the PShifter bars represent percentage energy savings of PShifter over UP. PShifter achieves energy savings of up to 22%, 16%, and 23% with geometric means of 10.5%, 5.3%, and 9.15%, for miniFE,

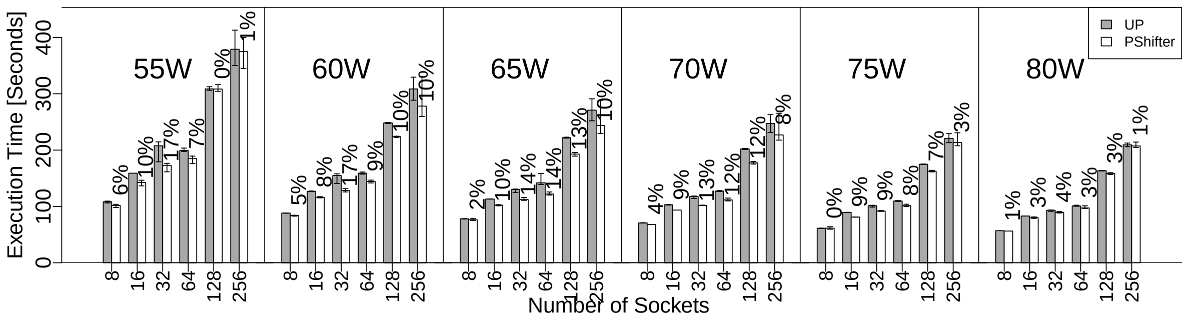


Figure 5.8 Runtime and % improvement of PShifter over UP for miniFE for job power = (Avg. Power per Socket) × #Sockets

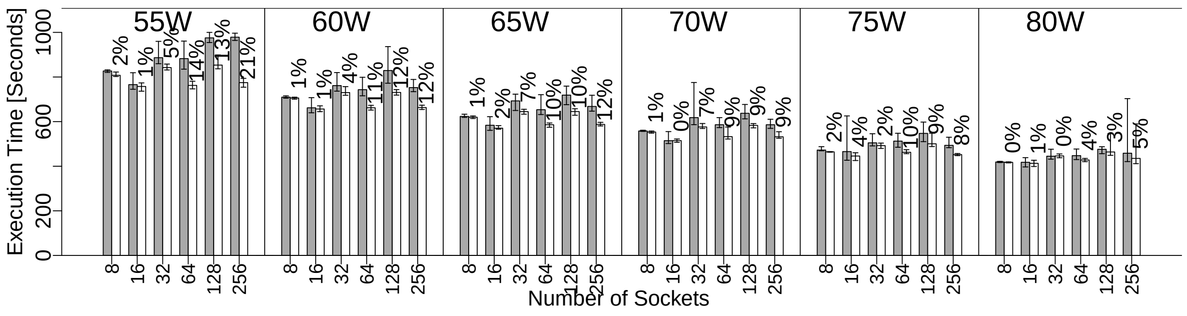


Figure 5.9 Runtime and % improvement of PShifter over UP for CoMD for job power = (Avg. Power per Socket) × #Sockets

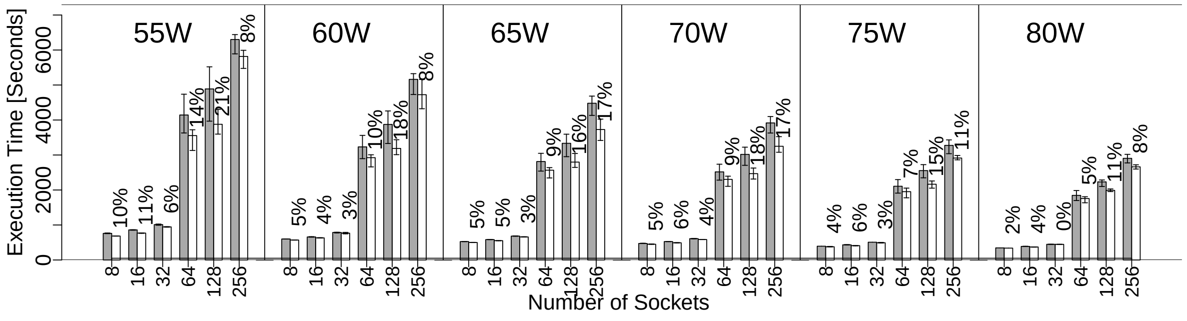


Figure 5.10 Runtime and % improvement of PShifter over UP for ParaDiS for job power = (Avg. Power per Socket) × #Sockets

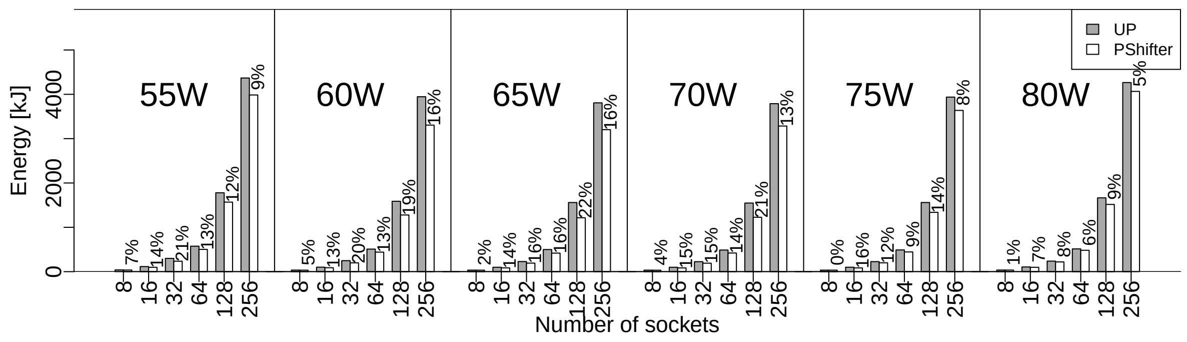


Figure 5.11 Energy and % improvement of PShifter over UP for miniFE, power budget = (Avg. Power per Socket) × #Sockets

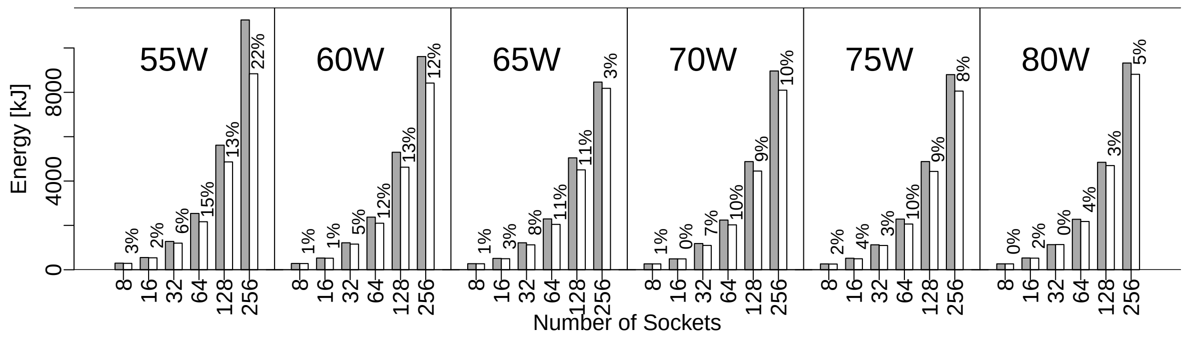


Figure 5.12 Energy and % improvement of PShifter over UP for CoMD, power budget = (Avg. Power per Socket) × #Sockets

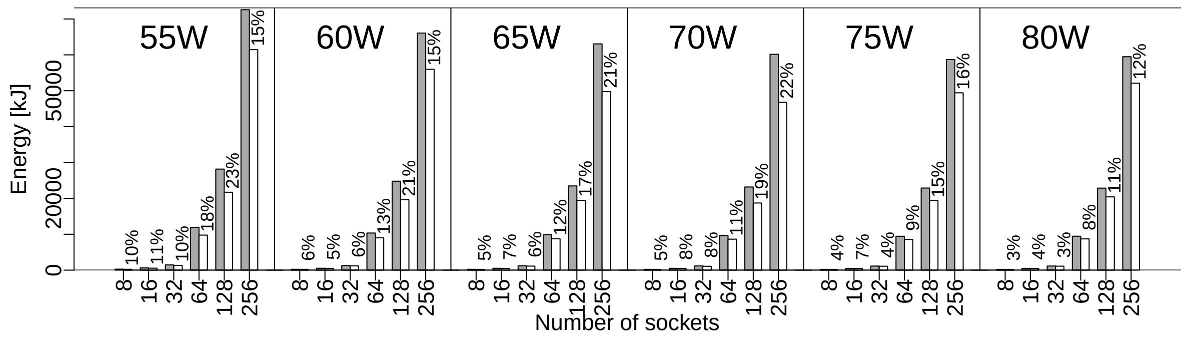


Figure 5.13 Energy and % improvement of PShifter over UP for ParaDiS, power budget = (Avg. Power per Socket) × #Sockets

CoMD, and ParaDiS, respectively. It is important to note here that even in case of 0% performance improvement for the 128 socket job of miniFE at an average socket power of 55W, PShifter achieves 12% energy savings, i.e., when PShifter is at par in performance, it may still gain energy savings over UP. PShifter never loses in energy either.

Observation 6: PShifter scales well with increasing socket count.

Under each power budget (Avg_Pow), PShifter achieves sustainable performance gains even at higher (64, 128, 256) socket counts. Exceptions to this observation are data for miniFE at lowest (55W) and higher (75-80W) power budget. The reasoning for the former case is discussed above. In cases of higher power budgets, it is observed that most of the overloaded sockets are already operating at maximum power (i.e., they are not power constrained) and, hence cannot benefit from additional power. PShifter shifts power away from underloaded to overloaded sockets but these overloaded sockets cannot consume any additional power but underloaded sockets now consume lower power. Hence, the job's energy consumption reduces (5-9%) but performance gains are not significant. This also explains why the performance gains of PShifter at the highest power budget are not as good as in cases of lower power budgets.

Observation 7: PShifter compliments application-specific load balancing and further improves application performance.

Some applications (e.g., ParaDiS) have a built-in application-specific load balancer that moves data from overloaded processes to underloaded ones to reduce the imbalance within a job. This leads to shorter wait times and thus better performance. Our experiments (Fig. 5.14) indicate when PShifter is combined with the application-specific load balancer, it leads to up to 6% and on an average 3.25%, 4.5% and 1% additional performance improvements on top of load balancing at 60W, 70W, and 80W average power per socket, respectively.

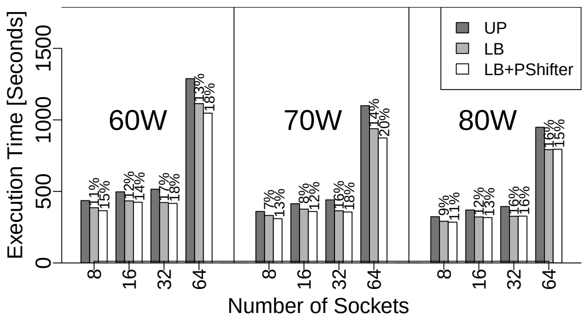


Figure 5.14 PShifter compliments application-specific load balancer for ParaDiS.

However, not all applications are equipped with an application-specific load balancer and developing one requires domain-specific knowledge and modifications to the application. PShifter avoids this developmental effort and provides a general system software solution that can be deployed across applications to reduce workload imbalance by using power as an actuator.

Dynamic Phase-change Detection and Power Management

Observation 8: PShifter detects phase changes at runtime and accordingly shifts power to minimize the new imbalance.

We illustrate the dynamic power management by PShifter in in two figures. Fig. 5.15 shows the performance imbalance within a 16 socket job for two consecutive phases (shown in the legend) of miniFE. The x-axis represents socket ID and the y-axis represents computation time for each socket per phase without our scheme. The sockets can be grouped into four groups, G1 (socket 1, 3, 5, and 7), G2 (socket 9, 11, 13, and 15), G3 (socket 10, 12, 14, and 16), and G4 (socket 2, 4, 6, and 8) in the increasing order of their computation times in the first phase. In the second phase, loads are reversed between groups, i.e., group G1 becomes the group of overloaded sockets compared to the other groups. G2 and G3 have more or less the same computational load while G4 becomes the group with the least load.

Fig. 5.16 shows the power profile for the same job as power decisions are made by PShifter during runtime. The x-axis shows the timeline in seconds, the left y-axis shows socket and the right one shows total job power. Power consumed by the 16 sockets of the job and job power are represented by the symbols shown in the legend. The job's power budget is set to 55W*16=880W. All the sockets are initially capped at 55W. As the execution progresses starting from Phase 1, sockets belonging to groups G1 and G2 start giving up power as they are underloaded sockets compared to the rest of the groups while sockets from groups G3 and G4 gradually gain more power in the order of their computational load. After the initial power shifting (from 0 to 7 seconds), socket power stabilizes for each socket until the phase change occurs. After the phase change at 33 secs, power is shifted from groups G2, G3, and G4 to G1 as it is the most overloaded group of sockets. Sockets from groups G2 and G3 stabilize in the range of 40W to 55W while G4 sockets stay at minimum power as they have the least load.

For each curve, you can see multiple data points that show drops in socket power followed by a steep rise in a periodical fashion. The drops are a result of local agent's invocations that lead to lowering of power caps in case of $PV < 0.95$. The rise in power caps is a result of cluster agent invocation that feeds back the unused power into the sockets. The total job power remains consistently below the job's power budget. With PShifter the job completes in 85 seconds.

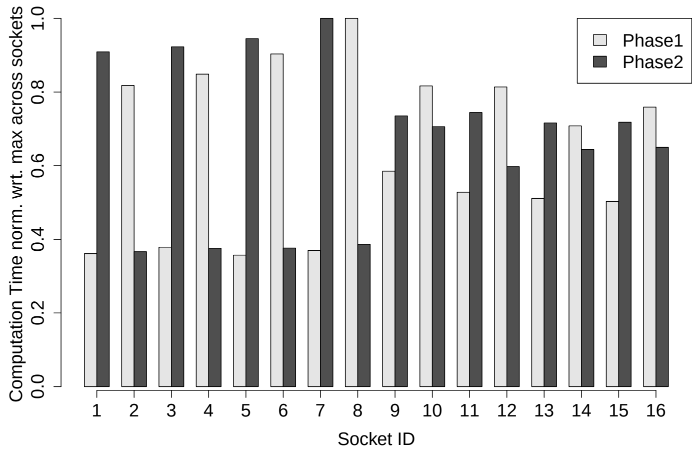


Figure 5.15 Imbalance in two phases of a 16 socket job

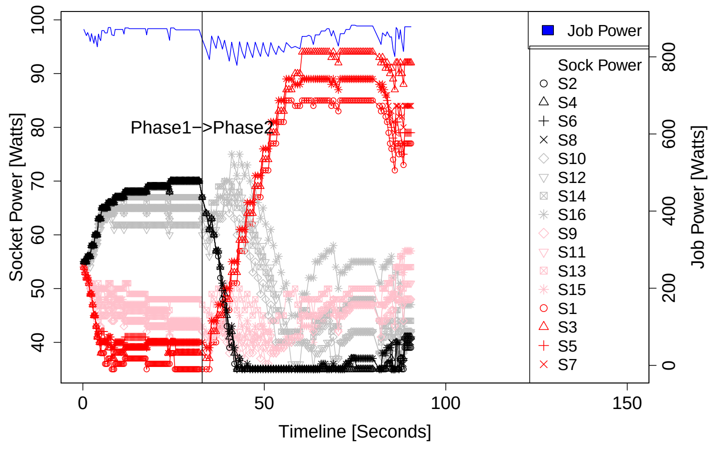


Figure 5.16 Power Profile for a 16 socket job with PShifter

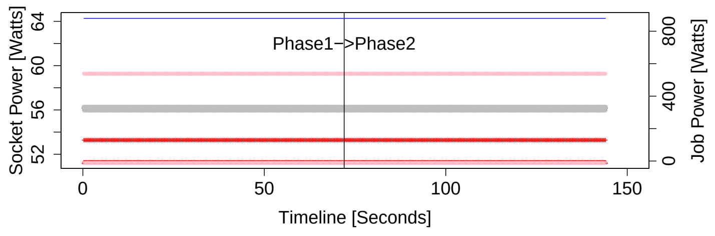


Figure 5.17 Power Profile for a 16 socket job with PTune

5.4.2 Comparison with PTune

In recent work, Gholkar et al. [Gho16] presented *PTune*, a process variation-aware power tuner that uses performance characterization data for all sockets on a cluster and application execution characteristics to minimize the runtime of a job under its power budget. PTune makes static (at the time of job scheduling) decisions for every job about the choice of sockets and the distribution of the job power budget across them. It eliminates inefficient sockets from the job allocation that are unaffordable under an assigned power budget and distributes power non-uniformly across the chosen sockets to counter the effect of performance variation across the cluster. To enforce its policy, PTune requires an application to be *moldable* (number of ranks is modifiable) as the number of nodes in the optimal configuration for a power budget varied based on availability of nodes and the power characteristics of their sockets.

While PTune shares the common objective of achieving performance improvement under a fixed job power budget, it is oblivious of the algorithmic or workload imbalance within a job. As PShifter makes measurement-driven dynamic power decisions at runtime, it detects such workload imbalance. Hence, unlike PTune, PShifter rebalances a job *with or without workload imbalance* on a non-homogeneous machine operating under a power constraint. In addition to this, PShifter easily compliments today's parallel computing system software. It is a stand-alone runtime system that can run alongside every job once it is scheduled by a conventional scheduler. It does not need to co-ordinate in any way with other modules of the system — unlike PTune, which needs to work with the conventional scheduler as it modifies the job allocation for power-constrained jobs. PTune also requires applications to be moldable, which they currently are not.

Table 5.2 compares the completion times of perfectly balanced miniFE jobs with the two power management schemes, PTune and PShifter, for 8, 16 and, 32 socket jobs at 55W per socket. PShifter achieves an average performance improvement of 9%, 12% and 22% over PTune for 8, 16, and 32 socket jobs, respectively.

Table 5.2 Completion time of MiniFE with PShifter and with prior work, Power Tuner (PTune)

Socket Count	PTune	PShifter
8	154s	139s
16	174s	153s
32	181s	141s

Unlike PTune, PShifter achieves these speedups without requiring any prior information about the sockets or the application's power-performance curves. It is important to note here that the data (completion time) for PTune does not take the training time (for characterization runs) into account.

It only represents the final completion time for the jobs once they are configured by PTune. Even though this gives PTune an advantage, PShifter outperforms PTune for balanced codes. Generation of training data would require additional time and energy for PTune that is not accounted for in Table 5.2.

PTune is oblivious of the runtime imbalances within a job and, hence, is not designed for optimization of load imbalanced jobs unlike PShifter. This is depicted is Fig. 5.17, which shows the power profile for PTune with static decisions running the same application as shown in Fig. 5.16 (same axes and power budget as before). At the job initialization, PTune caps sockets non-uniformly by taking the power and performance characteristics of the sockets into account using prior information. This power distribution configuration remains constant throughout the execution. With PTune, this job completes in 144 seconds (40% slower than with PShifter) as the power distribution is not aligned with the load distribution which can only be detected at runtime. The phase change is delayed compared to PShifter as shown in the figure. PTune neither detects nor responds to this phase change as it makes static decisions at job scheduling time unlike PShifter, which makes measurement-driven dynamic decisions.

5.4.3 Comparison with Conductor

Conductor is a dynamic scheme closest to PShifter with an on-line power-constrained runtime system [Mar15]. Conductor requires applications to be configured with one MPI task per RAPL domain (per processor or socket depending on the Intel chip), and to be OpenMP enabled since it exploits thread parallelism. This hard requirement for hybrid MPI+OpenMP limits Conductor to a subset workloads that excludes MiniFE, which supports MPI only, but not in conjunction with OpenMP. We therefore exclude MiniFE from this comparison.

Conductor speeds up an application's critical path through an adaptive socket power-allocation algorithm that periodically performs dynamic voltage frequency scaling (DVFS) and dynamic concurrency throttling (DCT) based on application behavior and power usage. Conductor consists of two main steps: (a) Configuration space exploration determines the best frequency and thread concurrency level for individual computation tasks under the power limit. (B) Power reallocation intelligently re-assigns power to the application's critical path.

A typical time step in an application may comprise several computational sections. Conductor selects the best configuration (concurrency level and DVFS state) per section. (1) Conductor records power and performance profiles per section for all possible configurations. To reduce the run-time overhead, Conductor performs this step in a distributed fashion by assigning a unique configuration subset to each MPI process and then gathers these profiles at the end of the time step. From these profiles, Conductor creates a list of of power-efficient configurations (that are *Pareto-efficient*) per code section and subsequently selects any new (lower power) configurations during execution.

(2) Conductor monitors power usage per MPI process, estimates the critical path of the application using historical data collected on-line, and reallocates power to speed up the critical path. In more detail, monitoring provides the means to reduce the power consumption on non-critical paths via a low-power configuration that finishes computation just in time without perturbing the critical path. This frees up some power. Differences between MPI processes and paths may be due to an application's load imbalance or differences in power efficiency between sockets. Conductor allocates more power to the processes on the critical path to speed up the application without violating the job power constraint. Conductor performs the power reallocation step at the end of several time steps demarcated using source-level annotations, which must be added by the user.

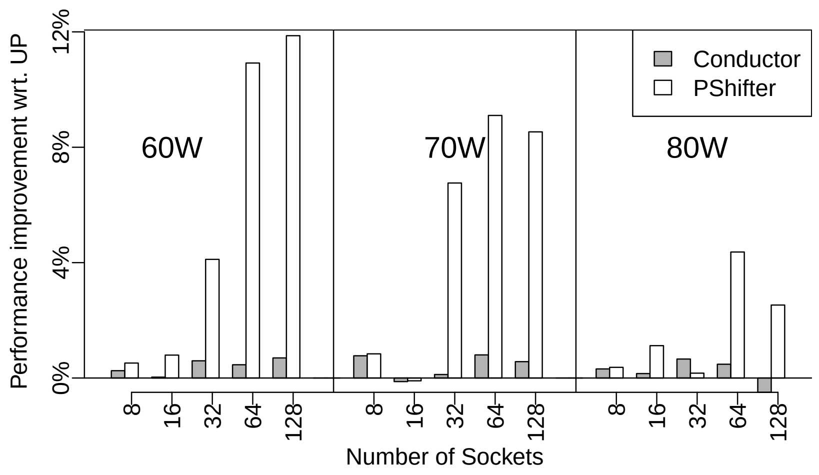


Figure 5.18 Comparison of PShifter with prior work, Conductor for CoMD.

Fig. 5.18 and Fig. 5.19 depict the performance improvement (y-axes) of PShifter and Conductor over Uniform Power for CoMD and ParaDiS, respectively, for different numbers of sockets (x-axes). We used one MPI process with up to 12 threads per socket. For both applications, Conductor consistently performs worse than PShifter. Conductor's lower performance is due to three reasons: (1) Conductor's heuristic for power re-allocation has limitations. The heuristic re-distributes unused job-level power to processors in the order of fraction of time spent near the processor power limit. But this results in inefficient power allocation compared to PShifter's power allocation because for long-running computation Conductor's heuristic depends on *average* power usage, which ignores spikes in power demands of the computation tasks. (2) Conductor's configuration selection phase relies on the repetitive nature of code in terms of the computational load and execution paths across processors. Since both the applications were load imbalanced across

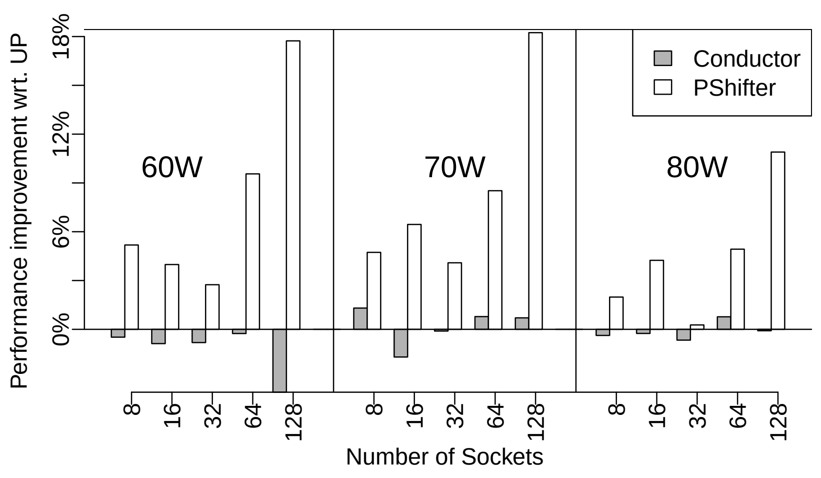


Figure 5.19 Comparison of PShifter with prior work, Conductor for ParaDiS.

processors (ParaDiS also being non-deterministic over time), Conductor was forced to perform configuration exploration *sequentially* per process, which contributed to the performance degradation, especially at lower operating frequencies and core counts. Although amortized over a large number of timesteps in our evaluation, this degradation in the configuration exploration step offset the performance gains in subsequent power re-allocation steps. (3) Conductor's heuristic for power management depends on algorithm-level knobs such as number of samples before re-allocating power, fraction of power donated/re-distributed among processors, and the power threshold to trigger power balancing. While it is easy to set up these knobs for repetitive, load-balanced applications, selecting a performance-optimizing combination of them for load-imbalanced applications results in exhaustive search. Our analysis showed that even the best-performing combination of these knobs resulted in thrashing between disjoint power schedules as the load imbalance changed over the run-time for ParaDiS. The performance degradation due to inefficient power schedules offset the performance gains of efficient power schedules. Table 5.3 summarizes the comparison of PShifter with the closest prior work.

5.5 Related Work

High performance computing (HPC) has increasingly been driven by power constraints in the past ten years. BlueGene/L was an early system originally based on an embedded processing core to limit power consumption. HPC facilities with DVFS have been studied for a long time. MPI applications were shown to often benefit by trading a slight increase in execution time (or sometimes even none when memory bound) due to running at lower frequencies for a significant reduction in

Table 5.3 Comparison of PShifter with PTune and Conductor

Feature	**PShifter**	**PTune**	**Conductor**
Power management	dynamic	static	dynamic
MPI	✓	✓	X
MPI+OpenMP	✓	✓	✓
Non-moldable jobs	✓	X	✓
Load-imbalance	✓	X	✓ low performance
No prior data required	✓	X	✓
Overheads	low	high	high
Deployment needs no modification to status-quo system	✓	X	✓

power [Fre05a]. DVFS has been combined with performance modeling and prediction to reduce the runtime of MPI codes under energy constraints [Spr06]. A plethora of algorithms exploit DVFS to save energy of HPC jobs [**4118669**; Lim06; Rou07; Rou09; Fre05b; Bai15; HF05; Ge07; Bha17]. Prior work [CM06; Li10] has also leveraged the effect of concurrency throttling and thread locality to save power and increase performance. While these approaches successfully lower the energy footprint of the jobs, they are ineffective in *enforcing caps* on job-level power budgets. Our work, on the other hand, uses power actuators via Intel's RAPL interface and guarantees that the job-level power constraint is never violated. While RAPL has been explored previously, PShifter is first (to our knowledge) to demonstrate that feedback-driven power reallocation transparently and without any training, results in effective computational load balancing. The novelty of the submission is the combination of the PID controller and RAPL for HPC applications, even across phase changes.

Early work exploited DVFS to reduce CPU frequencies during idle time, e.g., due to early arrival at MPI barriers and collectives [Kap05]. An ILP-based approach to model energy [Rou07] was demonstrated to be effective during the runtime of MPI codes to determine optimal power levels with little to no impact on execution time [Rou09]. Another effective method is to simply power down nodes when not needed to reduce energy [Pin01]. These works were trying to reduce power without sacrificing performance by much when utilizing just one core of a node resulting in underutilization of the system. PShifter differs in that its foremost objective is to guarantee a power constraint followed by trying to *not* degrade performance — and, as experiments showed, successfully so, as most runtimes are *reduced* — while utilizing all cores of a node.

An ILP-based runtime approach has been shown to determine how many cores an application should be run on to stay within a given power budget [Tot14; Lan15]. Our work differs from this work in terms of granularity and adaptivity. We manage power across resources at processor chip level exploiting dynamically adaptive feedback methods.

Capacity-improving schemes that increase job throughput have been developed under power limitations by exploiting "hardware overprovisioning", i.e., by deploying more nodes that will be powered at a time [FF05b; Pat13; Sar13a; Sar14; FF05a]. In such a system, the characteristics of a code under strong scaling were used to calculate the optimal number of processors considering core and memory power [Sar13b]. Just by exploiting DVFS at the granularity of a job, runtime and power can also be reduced [Eti10b; Eti10a]. Modifications to the batch scheduler in how small jobs are backfilled depending on their power profile can further increase job capacity [**patki2**]. These schemes do not deal with application imbalance. They, directly or indirectly, assign power budgets to the jobs running on the machine and assume a uniform power distribution within a job. Our solution compliments these approaches by dynamically detecting the imbalance and shifting power within a job to the resources where it is required with the objective of improving the job's performance. We do not explicitly aim at *maximizing* the machine's throughput, which is beyond the scope of this paper, but we often *improve* throughput as a side effect of PShifter. More specifically, each job finishes *early* under our scheme compared to the uniform power distribution scheme. This effectively improves the overall throughput of the system.

In other work, power was shifted within systems via scheduling while adhering to a global power cap [Ell15]. While this approach salvages the unused power, it does not detect wasteful power consumption of the processors at the barriers or other collectives resulting from any imbalance in the job. Our approach is able to reduce this waste and redirect power where it can be better utilized. Power balancing is a technique shown to be able to leverage differences in performance across a set of nodes and their cores [Ina15]. However, one pitfall of this method is its assumption, even though resulting in a good balance, that power and processor frequency have a proportional relationship. Later work indicated that power and processor frequency do not have a linear correlation opening up leverage for more refined power tuning [Gho16]. In a more recent work, waiting cores in the communication phase were power-gated and the saved power was then redirected to other active cores [**7573800**]. This work made an assumption that the time to power-gate and wake-up a processor is greater than the communication delay. First, PShifter does not rely on any such assumption. Second, with PShifter, sockets that are not on the critical path operate at lower power not just during communication but also during longer computation phases leading to significant power savings that can be used to accelerate the computation of other sockets on the critical path.

Our PShifter work is unique in that it neither makes any assumptions about the power-performance relationship of the processors nor does it require prior information about the variation across the processors. This, and the fact that it adapts dynamically to changes in execution behavior, are the biggest virtues of our scheme. PShifter makes measurement-based decisions by monitoring the state of the system. As this is a dynamic runtime system, it can also sense the imbalance resulting from the unequal division of work across processors in iterations in addition to the static imbalance induced due to performance variation across processors under power caps. Proposed frameworks like Redfish, the PowerAPI, and Intel's GEOPM [Red; Gra16; Api] can integrate PShifter as a unique closed-loop feedback-based policy for job power management on a cluster. PShifter also relieves the application developers of the burden to explicitly indicate phase changes as required by the APIs (like GEOPM) as PShifter automatically detects phases without any explicit information from the developer.

5.6 Summary

We presented PShifter, a feedback-based hierarchical solution for managing power of a job on a power-constrained system. PShifter makes dynamic decisions at runtime solely based on measurements. Unlike prior work, it does not depend on any a prior data about the application or the processors. At job level, PShifter employs a cluster agent that opportunistically improves the performance of a job while operating strictly under its power constraint. It does so by dynamically re-directing power to where it is needed. At processor level, PShifter employs local agents that aim to reduce the energy of the processors they manage. They achieve this by reducing the power of the processors that incur long wait times. Our evaluations show that PShifter achieves a performance improvement of up to 21% and energy savings of up to 23% compared to a naïve approach. Compared to a static power scheme, PShifter improves performance by up to 40% and 22% for codes with and without phase changes, respectively. Compared to a dynamic power scheme, it improves performance by up to 19%. PShifter transparently and automatically applies power capping non-uniformly across nodes in a dynamic manner adapting to changes during execution, simply by linking the PShifter library with or preloading it to an application.

CHAPTER

6

UNCORE POWER SCAVENGER: A RUNTIME FOR UNCORE POWER CONSERVATION ON HPC SYSTEM

One approach of achieving the goal of an exaflop under 20-30 MW power envelope was to develop solutions that improve the power efficiency (performance per Watt) of an HPC system by maximizing performance under a fixed power budget. Prior work presented in Chapters 3- 5 (Power Tuner, Power Partitioner and Power Shifter) focussed on this approach. Another direction of advancing toward this goal is to reduce the wasteful consumption of power that does not contribute to performance. In this work, we propose the Uncore Power Scavenger (UPS). UPS attempts to conserve power without significant impact on performance.

6.1 Uncore Frequency Scaling

From Haswell processors onward, Intel has introduced uncore frequency scaling (UFS) that can be used to modulate the frequency of the uncore independent of the core's operating state. UFS is exposed to software via a model-specific register (MSR) addressed at 0x620. Bits 15 through 8 of this MSR encode the minimum uncore frequency multiplier while bits 7 to 0 encode the maximum uncore frequency multiplier. The product (frequency multiplier x 100MHz) gives us the frequency.

For our architecture the minimum and the maximum uncore frequency multipliers are 12 and 27, respectively. We use the msr-safe [Sho14] kernel module to read from and write to the UFS MSR.

6.1.1 Single Socket Performance Analysis of UFS

We conducted our experiments on Broadwell nodes. Each node has two sockets. Each socket hosts one Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz processor. Each processor has 8 cores. Each node has 128GB memory. For performance analysis we present datasets for Embarrassingly Parallel (EP), the Block Tri-diagonal solver (BT) and the Multi-grid solver (MG) from the NAS parallel benchmark [Nas] suite. as they have distinct characteristics. EP is a compute-intensive benchmark, BT is a last level cache-bound benchmark and MG is a memory-bound benchmark. We used the MPI version of these codes in our experiments. The uncore frequency was modulated from 2.7GHz to 1.2GHz in steps of 0.1GHz. We report performance in terms of job completion time in seconds, average package power and DRAM power in Watts, memory bandwidth and LLC misses per second. Each experiment was repeated five times. We report averages across five runs.

Figures 6.1(a), 6.1(b) and 6.1(c) depict the effects of uncore frequency scaling on EP, BT and MG, respectively. The x-axis represents uncore frequency (from high to low). The default uncore frequency is 2.7GHz. Each figure presents five datasets, viz., completion time in seconds, package power and DRAM power reported by Intel's Running Average Power Limit (RAPL) [Int11] registers, memory bandwidth and LLC misses per second. The y-axis represents data normalized with respect to the default configuration (UF=2.7GHz). Table 6.1 shows the raw data for EP, BT and MG at 2.7GHz.

Table 6.1 Metrics for EP, BT and MG at 2.7GHz

Metric	**EP**	**BT**	**MG**
PKG power	40W	46W	50W
DRAM power	19W	23W	54W
Memory Bandwidth [mbps]	8	3983	37327
LLC misses per second	0.003M	3M	33M

Observation 1: UFS leads to (package) power savings without significant performance degradation for CPU-bound applications.

This is illustrated by Figure 6.1(a) showing data for EP, a representative CPU-bound benchmark. EP uses L1 and L2 caches aggressively but otherwise has a negligible memory footprint. For EP, package power decreases linearly with the uncore frequency. However, completion time and DRAM power remain constant. This is because EP neither uses the LLC nor the memory controllers heavily. The reduction in package power can be attributed to the uncore's idle power consumption.

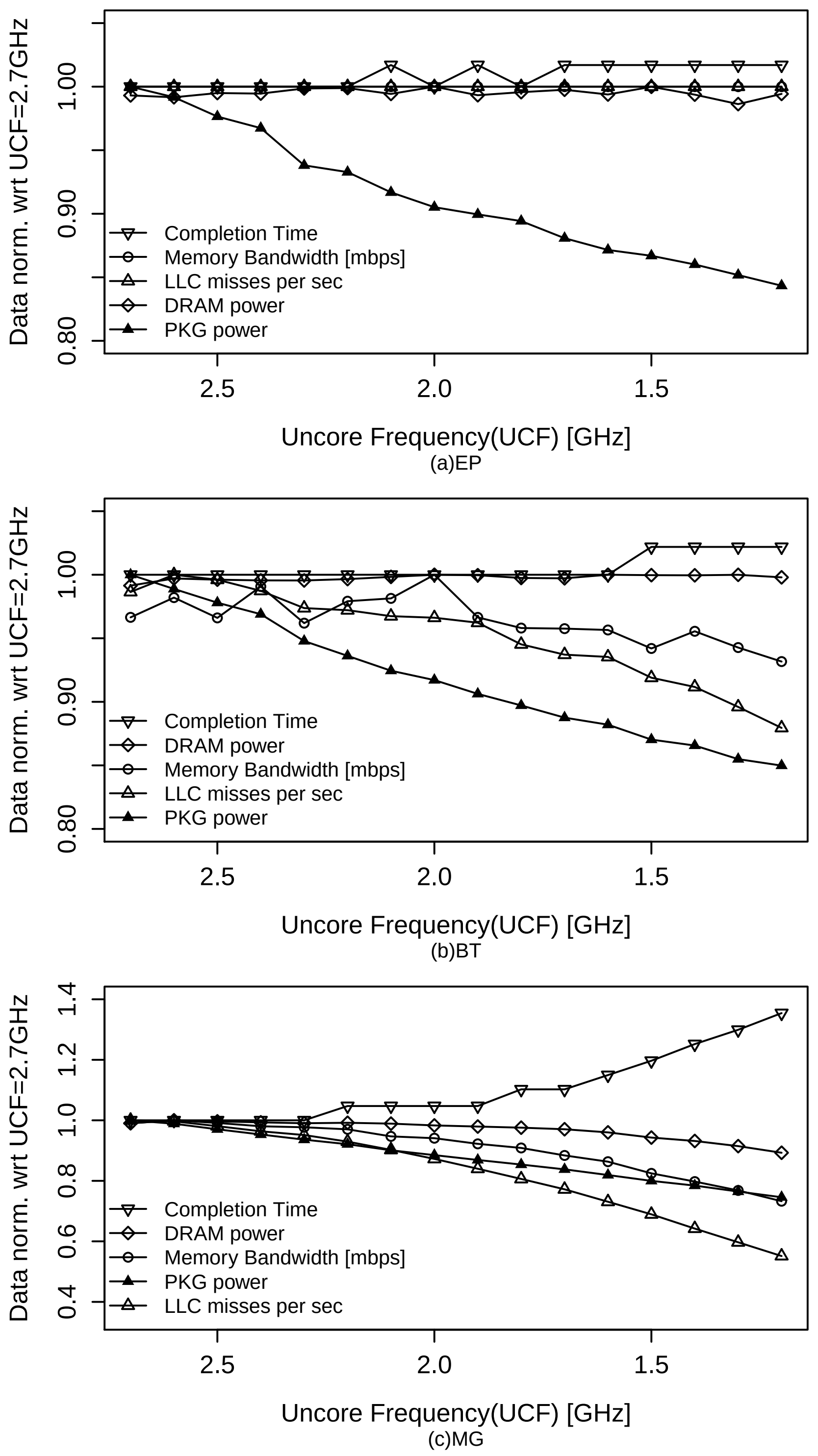


Figure 6.1 Effects of UFS on EP, BT and MG.

Observation 2: UFS reduces power for cache-bound applications but aggressive frequency reduction may lead to performance degradation.

This is illustrated by Figure 6.1(b) showing data for BT, a cache-bound applications. It uses LLC aggressively. With the default configuration, the LLC cache miss rate is only 3M per second as most of its LLC accesses are hits and memory bandwidth consumption is 3.9GB/s. Package power decreases linearly with the uncore frequency. However, completion times increase slightly at lower uncore frequencies. Reducing the uncore frequency leads to fewer LLC misses per second and lower memory bandwidth which leads to nominal performance degradation (slowdown) only beyond 1.5GHz. It is important to note that the DRAM power remains constant.

Observation 3: UFS leads to significant performance degradation for memory-bound applications.

This is illustrated by Figure 6.1(c) showing data for MG, a memory-bound application. With the default configuration, its LLC miss rate is 33M per second while it consumes 37GB/s ($\approx$10X that of BT) of memory bandwidth. Consistent with previous observations, even for MG package power decreases linearly with the uncore frequency. The completion time increases linearly with the uncore frequency under 2.1GHz. Reducing the uncore frequency leads to slower LLC and slower memory controllers. Slower memory controllers send memory requests at a lower rate to DRAM leading to lower DRAM power consumption as shown in the figure. Unlike EP and BT, DRAM power reduces with the uncore frequency beyond 2.1GHz for MG. This decline in DRAM power is also coupled with significant performance degradation beyond 2.1GHz reaffirming our previously stated inference.

From observations 1-3, we conclude the following:

- *The decline in DRAM power as a result of lowering the uncore frequency is an indication of potential significant performance degradation.*
- *The uncore frequency can be reduced safely without a significant impact on performance only up to the point until which the DRAM power remains unchanged.*

Observation 4: Applications consist of multiple distinct phases spanning across the spectrum from compute-intensive to memory-intensive.

This is illustrated by Figure 6.2 depicting the DRAM power profile and the IPC profile of MiniAMR. The x-axis represents the timeline while the y-axis represents data normalized with respect to the maximum data points. Five distinctly identifiable phases, P1-P5, are annotated in the plot. P1-P5 are short compute-intensive phases characterized by low DRAM power and high IPC occurring between relatively long memory-intensive phases characterized by high DRAM power and low IPC. Hence, the phase transitions can be detected as follows:

- The transition from a compute-intensive phase to a memory-intensive phase can be identified by a rise in DRAM power.

- The transition from a memory-intensive phase to a compute-intensive phase can be identified by a decline in DRAM power.

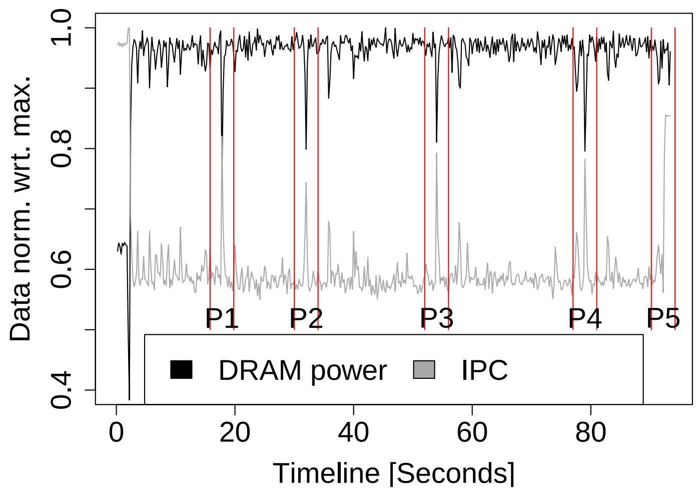


Figure 6.2 Phases in MiniAMR

Phase Change Detection

Let ΔIPC be the change in IPC and $\Delta DRAM\_power$ be the change in DRAM power during a fixed time interval. Algorithm 2 depicts the pseudo code for phase change detection.

Algorithm 2 Phase Change Detection

```
1: procedure DETECT_PHASE_CHANGE
2:
3:     if $\Delta DRAM\_power == 0$ then
4:         no phase change detected
5:     else if $\Delta DRAM\_power > 0$ then
6:         phase change detected
7:         compute-intensive to memory-intensive
8:     else                                        ▷ $\Delta DRAM\_power < 0$
9:         phase change detected
10:        memory-intensive to compute-intensive
11:    end if
```

Hypothesis: Static UFS is sub-optimal for applications with distinct phases.

By combining observations 1-4 we hypothesize that for an application consisting of multiple phases with distinct uncore utilization, DRAM access rates and CPU intensities, statically setting single uncore frequency for entire execution is sub-optimal. Hence, we need a dynamic approach that automatically detects phase changes and determines the best uncore frequency for each phase.

While techniques like power gating can be leveraged to save power for phases with zero uncore utilization [Mei11; GK06; MW12; Sam13], i.e., while all processes are suspended, uncore frequency scaling achieves power conservation during phases with non-zero but low uncore utilization, which is often the case for HPC applications. Using techniques like sleep states requires prediction of the length of the zero utilization phases to evaluate the trade-offs of the overheads of entering and exiting from a sleep state. Uncore frequency scaling is free of such overheads. Hence, we use uncore frequency scaling in our proposed approach.

6.2 Uncore Power Scavenger (UPS)

We propose UPS, a power-aware runtime system that aims at conserving power by dialing down the uncore's operating frequency opportunistically without significant performance degradation. Figure 6.3 depicts the high-level architecture of UPS and the mapping of its components to the components of an HPC system. Figure 6.3(a) shows the architecture of a typical HPC cluster. It consists of multiple server nodes. Each server node hosts two sockets with a single 12-core processor. Multiple parallel jobs spanning across one or more nodes can run on a cluster, e.g., job1 runs on the top 4 nodes while job2 runs on bottom 4 ones. An instance of UPS runs alongside each job.

The high-level design of the UPS runtime is depicted in Figure 6.3(b). The UPS runtime system allocates one UPS agent to each socket in a job. Each UPS agent monitors its socket's performance and DRAM power. It dynamically and automatically manages a socket's uncore frequency to conserve power. UPS agents make asynchronous decisions based only on the local information pertaining to their respective sockets. Hence, UPS does not require any global synchronization across the individual UPS agents. The UPS agent is a closed-loop feedback controller [SC]. The general idea of a closed-loop feedback controller is depicted in Figure 6.4.

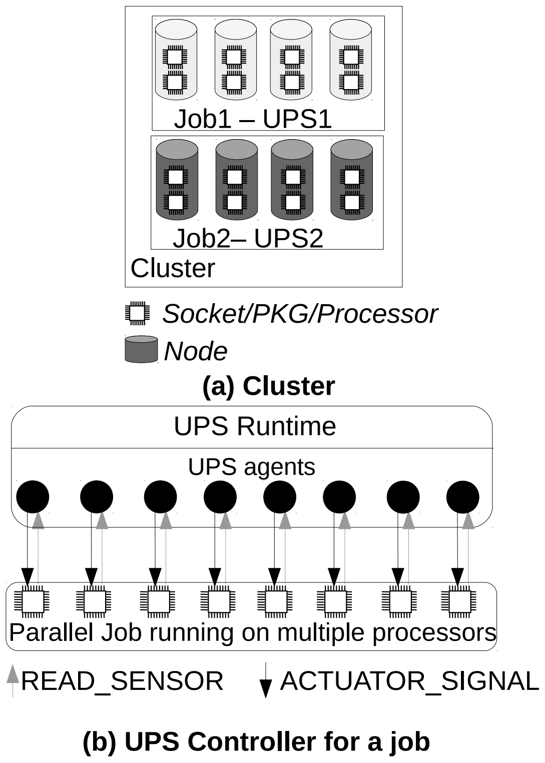


Figure 6.3 UPS Overview

A feedback-control loop consists of three stages that are repeated periodically, viz., sensor, control signal calculator, and actuator.

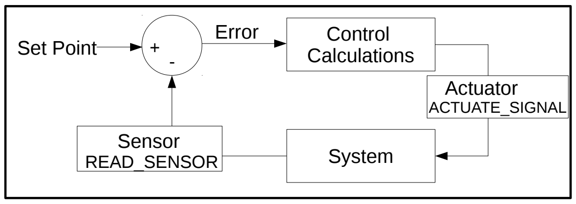


Figure 6.4 Closed-loop Feedback Controller

System is the component whose characteristics are to be monitored and actuated by the controller. The current state of the system is defined in terms of the *process variable* (PV). The desired state of the system, PV=K, where K is a constant, is called the setpoint (SP). It is input to the feedback controller. A feedback-control consists of three main modules, viz., sensor, control signal calculator, and actuator.

Sensor

The sensor periodically monitors the state of the system defined in terms of process variable. This is indicated by READ_SENSOR.

Control Signal Calculator

The error (err) in the system state is calculated as the difference between the setpoint and the measured process variable. The feedback controller determines the new value for the actuator as a function of this error to drive the system closer to the setpoint.

Actuator

The actuator applies the calculated signal to the system. This is indicated by ACTUATE_SIGNAL.

6.2.1 UPS Agent

Each UPS agent is a closed-loop feedback controller that performs three tasks. First, it periodically monitors the socket by measuring its performance in terms of instructions per cycle (IPC) and its DRAM power consumption. Second, dependending on the measured power and IPC, it determines the uncore frequency that would drive the system closer to the setpoint and actuates. Third, it detects phase changes at runtime and resets the setpoint for every new phase. The architecture of a UPS agent is depicted in Figure 6.5.

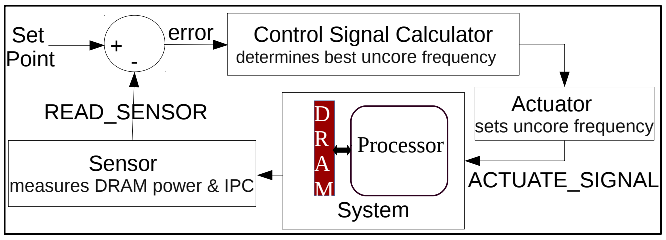


Figure 6.5 UPS Agent

System, Process Variable and Setpoint

UPS agent's system is a socket consisting of a processor and its local DRAM. The process variable of a UPS agent is its socket's DRAM power consumption, i.e., PV = DRAM power, measure of socket's DRAM utilization, which again, correlates with the uncore utilization. Prior work [GK06; Mei11; WM08] has used models that rely on multiple performance counter to determine application characteristics. This approach has higher overhead. UPS relies on a single register (DRAM RAPL MSR) and hence, issues just one MSR request to quantify/determine the state of the system. This has very low overhead. UPS automatically determines the setpoint, which is the maximum DRAM power consumption for every phase of the application, i.e., For an application with multiple phases, UPS agent automatically resets the setpoint on detecting a phase change.

Sensor

The sensor periodically monitors DRAM power and IPC of the socket. This is indicated by READ_SENSOR.

Control Signal Calculator

The control signal calculator takes the socket's DRAM power consumption and IPC measured at runtime as inputs. It then determines the current state of the system to decide the next course of action. Table 6.2 describes the variables used by the control signal calculator. Figure 6.6 presents the control logic.

Table 6.2 Variables in Control Signal Calculation

Variable	Description
ucf	uncore frequency
PV_DRAM_power	DRAM power consumption measured during current sampling interval
SP_DRAM_power	setpoint
err = $\Delta DRAMpower$	PV_DRAM_power - SP_DRAM_power
previous_IPC	IPC measured during previous sampling interval
current_IPC	IPC measured during current sampling interval
ΔIPC	current_IPC - previous_IPC
MAX_UCF	maximum uncore frequency for the architecture

During its first invocation (INITIALIZATION), the UPS agent sets the setpoint, SP_DRAM_power, to the observed DRAM power, PV_DRAM_power. After initialization, the system enters into a periodic control loop. IPC and DRAM power are measured for each sampling interval. ΔDRAM_power is calculated as $PV\_DRAM\_power - SP\_DRAM\_power$.

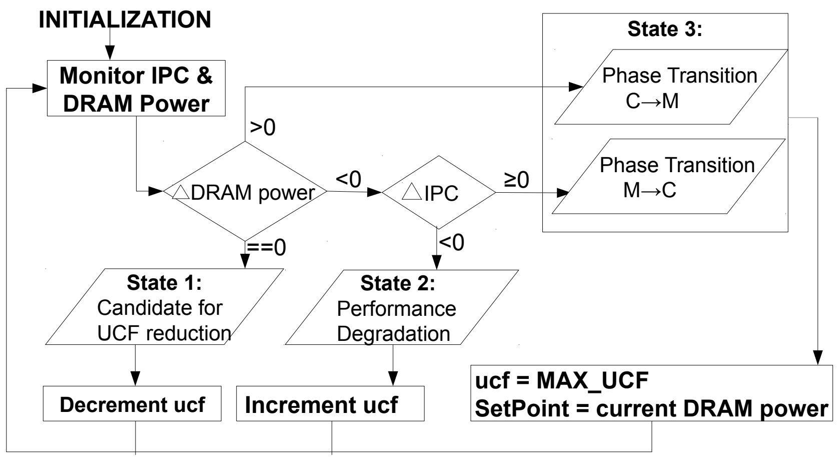


Figure 6.6 Control Logic: A State Machine

State 1: Candidate for UCF reduction

If ∆DRAM_power is zero, it implies that the DRAM power has not declined (from the setpoint for the current phase) as a result of the previous actuation. This puts the system in state 1, i.e., it is a candidate for lowering uncore frequency in the current iteration of the control loop. Therefore, the actuator decrements the uncore frequency and control returns to the monitoring block.

State 2: Performance Degradation

If ∆DRAM_power is less than zero, it is an indication of a drop in DRAM power from the setpoint for the current phase. This drop could either be an indication of a phase transition from a memory-intensive to a compute-intensive phase or a detrimental effect of excessive uncore frequency lowering in the previous iteration of the control loop. To identify the cause of the observed drop, we monitor IPC. ∆IPC is the difference between IPC during the current and the previous sampling intervals. If ∆IPC is less than zero, it is an indication of performance degradation. Hence, this observation puts the system in state 2. In response to this state, the actuator increments the uncore frequency and control returns to the monitoring block.

State 3: Phase Transitions

If ∆DRAM_power is less than zero and ∆IPC is greater than or equal to zero, it is an indication of a rise in CPU activity with a decline in DRAM power. This implies that a phase transition from memory-intensive to compute-intensive phase occurred. Hence, this observation puts the system in state 3 (3a).

If ∆DRAM_power greater than zero, it implies that the measured DRAM power is higher than the setpoint (maximum) for the current phase. This implies that there has a phase transition from compute-intensive to memory-intensive phase. Hence, this observation puts the system in state 3 (3b).

Actuator

The actuator is responsible for taking action in response to the determined state of the sytem. This is indicated by ACTUATE_SIGNAL. The UPS agent sets the socket's uncore frequency by writing to the MSR (at 0x620).

State 1: The actuator decrements the uncore frequency to conserve power.

State 2: The actuator increments the uncore frequency to revert the detrimental effects of the previous actuation.

State 3: On detecting a transition to a different phase, the actuator sets the uncore frequency to the maximum value (MAX_UCF) and sets the setpoint (say SP_DRAM_power') to the measured DRAM power (PV_DRAM_power).

Resetting the Setpoint to maximum DRAM power for a new phase: It is important to note here that in response to both the phase transitions (even in case of M->C) we set the uncore frequency to MAX_UCF. As a result of this, the socket is guaranteed to draw maximum DRAM power (at MAX_UCF) during the subsequent sampling interval of the new phase. If this maximum DRAM power (measured as PV_DRAM_power during the subsequent sampling interval) is greater than SP_DRAM_power', the system enters state 3 again. In this case, SP_DRAM_power is now intentionally set to the maximum DRAM power for the new phase.

After actuation, the control returns to the monitoring block of the control logic.

Sample Interval

We invoke the UPS agents periodically every 200ms in our experiments. We conducted a sensitivity study by varying the sample interval from 200ms to 1s. We observed that our runtime system has negligible overheads at 200ms while a longer sampling interval makes it sluggish because of two reasons. First, it has fewer opportunities of detecting phase changes and modulating the uncore frequency. Second, it leads to longer reaction times. A sampling interval of 200ms is sufficiently long to actuate the uncore frequency modulation and observe its effect and it is sufficiently frequent to have a fast reacting runtime system.

6.3 Implementation and Experimental Framework

We conducted our experiments on a cluster of 17 Broadwell nodes. Each node has two Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz processors and each processor has 8 cores. Typical HPC systems, like ours, provide exclusive node access. Each MPI job runs on uniquely allocated nodes. No two jobs (or applications) share nodes. We use all the cores of every socket in a job. We pin one MPI rank to a unique core on every socket. Typical MPI applications are NUMA-aware, i.e., all the cores access memory on the socket-local DRAM via the local memory channel consuming the local memory bandwidth. We observed that the remote memory bandwidth consumption is negligible and constant over time compared to the local memory bandwidth for all applications. Hence, we can use each socket's DRAM power as an approximation of the socket's uncore utilization.

UPS is a library that can be linked with the application at the time of compilation. It is implemented leveraging the MPI standard profiling interface [MG97] (PMPI). The UPS agents are initiated in the wrapper function of the MPI_Init MPI call, which is the first MPI function call invoked by an MPI application. We use signals to set timers for periodic invocation of UPS agents on every socket in a job. The system in restored to the default state (maximum uncore frequency multiplier is set to 27) in the wrapper function of the MPI_Finalize MPI call, which is the final MPI call invoked by an MPI application indicating its completion. As UPS is a shared library that can be linked to the application and it leverages the PMPI interface for intialization and termination, no modifications to the application's source code are required to use UPS. Applications only need to be linked to our library by modifying their makefile at the time of compilation.

At each invocation, a UPS agent periodically monitors the state of the system by measuring DRAM power consumption and IPC. For the DRAM power measurement, we leverage Intel's Running Average Power Limit (RAPL) [Int11]. From Sandy Bridge processors onward, Intel supports this interface that allows the programmer to measure (or constrain) the power consumption of a processor/socket/package (PKG) as well as the DRAM by reading from (or writing to) the RAPL MSR. We also use RAPL in some of our experiments to constrain the power consumption of the processors. RAPL has a power capping mechanism implemented in hardware that guarantees that the power consumption does not exceed the power limit specified by the user. The UPS agent measures IPC by reading the fixed counters [Int11].

Depending on the control calculations, the UPS agent modulates the uncore frequency by writing to the bits 0 to 7 of the UFS MSR addressed at 0X620 that encode the maximum uncore frequency multiplier. The minimum uncore frequency multiple is set to 12, the default value. The msr-safe kernel module installed on this cluster enabled us to read from and to write to Intel's RAPL and UFS MSRs and fixed counters indirectly via the libmsr library [Sho14].

In Figure 6.6, we compare ΔDRAM_power and ΔIPC to absolute zero. However, in the implementation we use an error margin of $\pm 5\%$ to account for minor fluctuations in the measurements.

UPS agents in a job do not need any global synchronization or communication with other agents. They monitor their respective sockets and make local decisions based on local knowledge about the system. Hence, UPS is a very light weight runtime system with only nominal overheads of reading and writing to MSRs that are several orders of magnitude lower than the total completion time of a typical HPC job.

6.4 Experimental Evaluation on a Multi-node Cluster

For evaluation we used Embarrassingly Parallel (EP), Scalar Penta-diagonal solver (SP), Block Tri-diagonal solver (BT), Multi-Grid (MG) and Fourier Transform (FT) from the NAS parallel benchmark suite [Nas] and CoMD and MiniAMR from the ECP proxy applications suite [Ecp]. We used the MPI version of these codes in our experiments. Our selection of applications ensures that we evaluate UPS with applications spanning across the spectrum from compute-intensive to memory-intensive. For example, EP is a CPU-bound application that uses uncore minimally, SP and BT use uncore (LLC) intensively with moderate memory (DRAM) access rates while other applications have high memory (DRAM) access rates. We report average power savings over the duration of the job and slowdowns caused by the UPS runtime system with respect to the baseline. We chose the default configuration, where Intel's firmware modulates the uncore frequency, as the baseline. We also compare an application's performance with UPS to its performance with Intel's Running Average Power Limit (RAPL) with equal power consumption. Each experiment was repeated five times. We report averages across all repetitions with standard deviation of 1-5%.

Figure 6.7 depicts power savings achieved with UPS, resulting slowdowns and energy savings with respect to the baseline. We show results for 64, 128 and 256 core experiments as shown in the legend. X-axes represent the benchmarks and applications. Y-axes represent (top to bottom) package and DRAM power savings, slowdown and energy saving with respect to the baseline.

Observation 1: UPS achieves up to 11% package power savings.
For applications like EP, MiniAMR, BT and SP that have longer CPU-bound phases than others (such as MG, FT, CoMD), UPS achieves higher package power savings. These codes keep the cores busy and incur fewer LLC misses per second. As a result, they are not as dependent on long latency memory accesses and the uncore as the rest. Hence, the uncore frequency can be lowered for the CPU-intensive phases to save power.

Observation 2: UPS has a marginal impact on DRAM power.
This is a desired effect. UPS is designed to reduce uncore frequency so long as it does not impact DRAM utilization. We measure DRAM utilization in terms of DRAM power. Hence, a substantial decline in DRAM power is an indication of decline in DRAM utilization (unless a phase change occurs) which can lead to performance degradation.

Observation 3: UPS causes worst-case slowdown of 5.5%.
In some cases, UPS achieves power savings at the cost of marginal slowdown for a few applications. This includes the overhead of the runtime system, i.e., measuring DRAM and PKG power, IPC, calculating the feedback and setting appropriate uncore frequency. The observed performance degradation is under 5.5% across all codes with some codes experiencing no slowdown.

Observation 4: UPS achieves up to 14% energy savings. UPS achieves energy savings of at least 5% across all applications and core counts, despite the slowdown incurred in some cases. UPS conserves energy by simply reducing the uncore's energy usage when it is underutilized, thus reducing energy wastage and shifting towards more energy efficient execution.

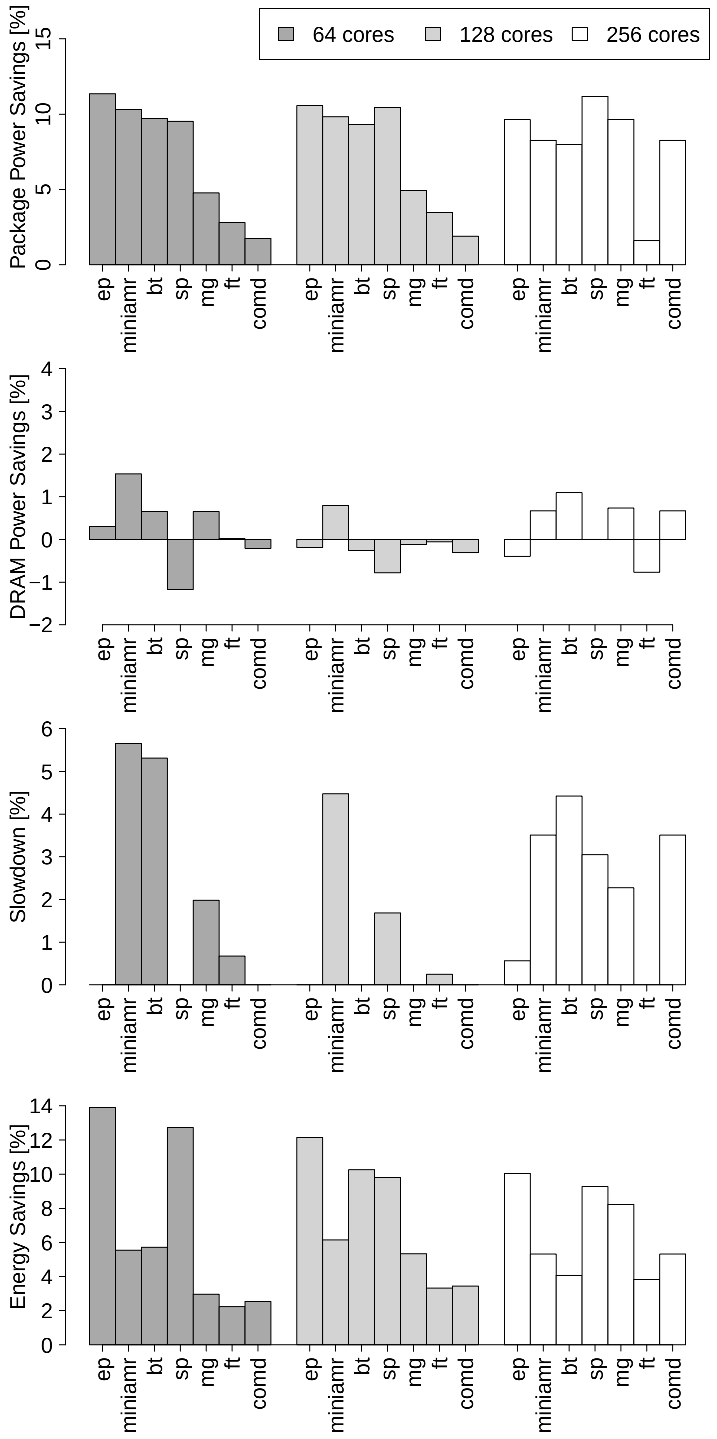


Figure 6.7 Package and DRAM power savings achieved with UPS and the resulting slowdowns and energy savings with respect to the baseline

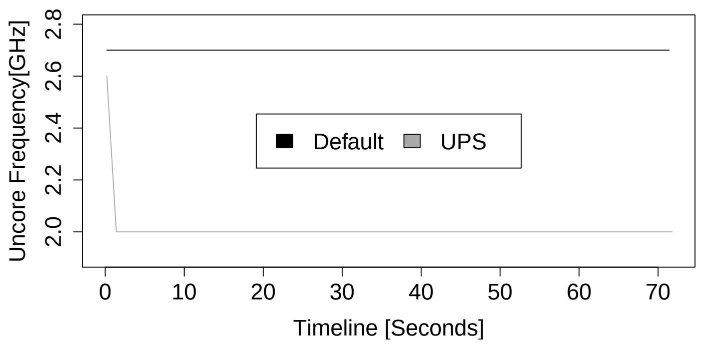


Figure 6.8 Uncore frequency profiles for EP with UPS and the default configuration.

Observation 5: Intel's default scheme consists of a naive algorithm that sets the uncore frequency to its maximum.
Intel's default algorithm sets uncore frequency to 2.7GHz on detecting even the slightest uncore activity. Figure 6.8 depicts the uncore frequency profile for runs of EP with default configuration (top) and UPS (bottom). X-axes represent the timeline while y-axes represent the uncore frequency. EP is a CPU-intensive embarrassingly parallel benchmark that does not use the uncore at all. It operates on core-local cache and incurs nominal LLC misses per second. Even in case of such a workload, the default scheme sets the uncore frequency to 2.7GHz, which is the maximum for our specific architecture.

Unlike this scheme, UPS intelligently modulates uncore frequency by making measurement-driven decisions at runtime based on DRAM power and IPC achieved by the socket. Our automatic and dynamic algorithm leads to a constant uncore frequency of 2GHz for the entire duration of EP. For other applications like CoMD and MiniAMR, the uncore frequency continuously changes.

Figure 6.9 and 6.10 depict job profiles of MiniAMR and CoMD, respectively, for 4 node (8 socket) runs. We show two sets of plots (left and right). Plots on the left are the job profiles with the default configuration (baseline) while those on the right are with UPS. X-axes represent the timeline while y-axes represent the uncore frequency, DRAM power, package power (top to bottom) for each of the sockets of the job. Each socket is represented by a unique color as depicted in the legend. With the default configuration, all sockets operate at the maximum uncore frequency (2.7GHz) constantly for both MiniAMR and CoMD. Hence, the uncore frequency profiles of all the sockets coincide. However, with UPS, the UPS agents dynamically modulate the uncore frequency of sockets.

Observation 6: UPS detects phase change and increases the uncore frequency to UCF_MAX leading to spikes in package power.
UPS detects phase changes based on deflection in DRAM power and IPC. Figure 6.9 shows the DRAM power profile for MiniAMR. At least five phase changes indicated by dips in DRAM power

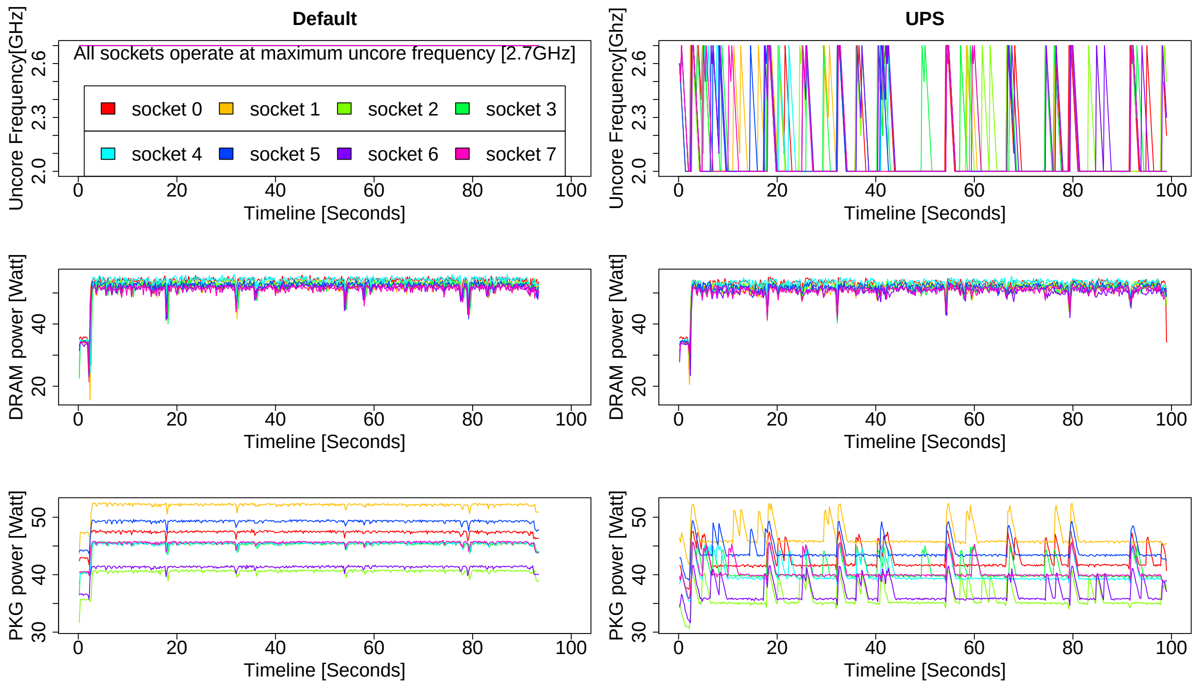


Figure 6.9 Power and uncore frequency profiles for 8 socket runs of MiniAMR with default configuration (left) and UPS (right).

can be observed. When each of these phase changes occurs, UPS agents automatically detect it and reset the uncore frequency of their corresponding sockets to UCF_MAX. This leads to a spike in the package power consumption as the uncore consumes higher power when operating at a higher uncore frequency. Even with the default configuration, the power consumption varies across packages in a job. This is a result of manufacturing variation across processors [Gho16; Ina15].

Observation 7: With UPS, packages spend more time at lower power than with the default configuration.

After the initial spike in package power for each newly detected phase, UPS gradually lowers the unore frequency to the lowest level before it starts affecting DRAM power. At this low uncore frequency, the packages consume less power than at UCF_MAX (depicted in PKG power plots in Figure 6.9 and Figure 6.10), thus leading to power savings.

Observation 8: Even in case of codes with frequent phase changes, UPS saves power.

For applications such as CoMD depicted in Figure 6.10 that incur frequent phase changes with short-lived phases, UPS does not stabilize at a constant lower uncore frequency for an extended duration. Instead, it intelligently and automatically modulates the uncore frequency between several (higher and lower) levels. As a result, packages spend some time at frequencies lower than that selected by the default scheme. This leads to overall power savings over baseline as shown in Figure 6.7.

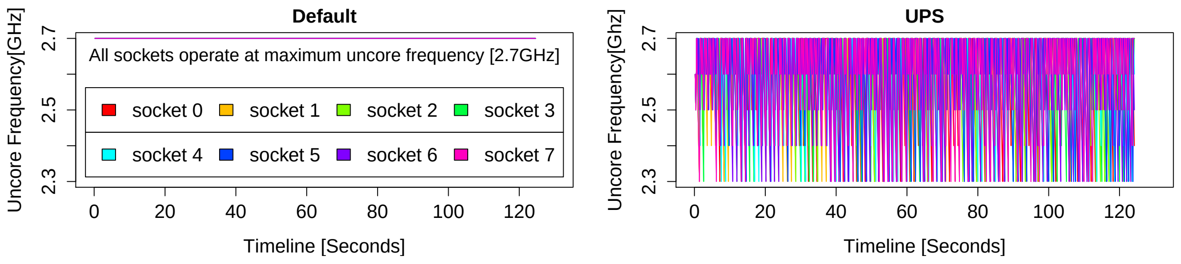


Figure 6.10 Power and uncore frequency profiles for 8 socket runs of CoMD with default configuration (left) and UPS (right).

Observation 9: UPS outperforms Intel's RAPL.
Intel's RAPL is a power capping mechanism implemented in hardware that restricts the power consumption of a processor to a value specified in the RAPL MSR. Figure 6.11 depicts the comparison of UPS with RAPL. We first recorded the completion time and the resulting power consumption (say P Watts) of an application with UPS. We then measured the application's performance without UPS under a P Watt power bound enforced by restricting power consumption of the processors in the job using RAPL. Figure 6.11 depicts the total (package and DRAM) power savings obtained by UPS, resulting performance improvements and energy savings with respect to RAPL. We show results for 64, 128 and 256 core experiments (see legend). X-axes represent the benchmarks and applications. Y-axes represent (top to bottom) total power savings, speedup and energy saving with respect to RAPL. The power consumption with UPS is within ±2% of RAPL. Interestingly, UPS achieves up to 20% speedup over RAPL leading to proportional energy savings. This indicates that even for a power-constrained computing paradigm that aims at maximizing performance under a strict power budget, UPS outperforms RAPL.

Observation 10: UPS deducts power from the uncore while Intel's RAPL deducts power from the cores to stay within the power limit.
This is illustrated in Figure 6.12, which depicts the effective per core frequency achieved with RAPL and with UPS for equivalent power consumption of BT. We first recorded the effective core frequency measured via Intel's APERF and MPERF MSRs [Int11] and the resulting power consumption (say P Watts) of an application with UPS. The effective frequency is calculated as $\frac{\Delta APERF * BASE\_FREQUENCY}{\Delta MPERF}$, where the base frequency on our architecture is 2.1GHz. We then measured the effective core frequency without UPS under a P Watt power bound enforced by restricting power consumption of the processors in the job using RAPL. The x-axis represents the timeline while the y-axis represents the effective core frequency.

RAPL lowers the core frequency from 2.3GHz down to 1.8GHz to save power, which leads to performance degradation. UPS achieves equivalent power savings by opportunistically reducing the uncore frequencies without affecting the core frequencies allowing the cores to operate at the

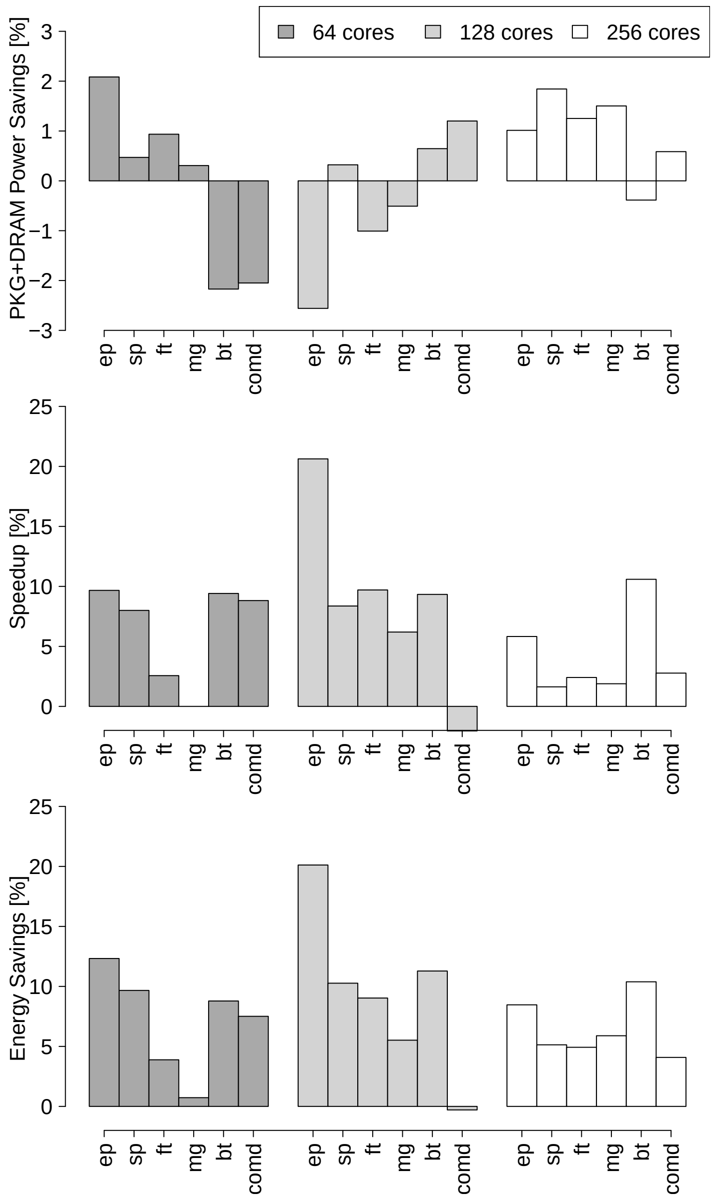


Figure 6.11 Package and DRAM power savings, speedups and energy savings achieved by UPS with respect to RAPL

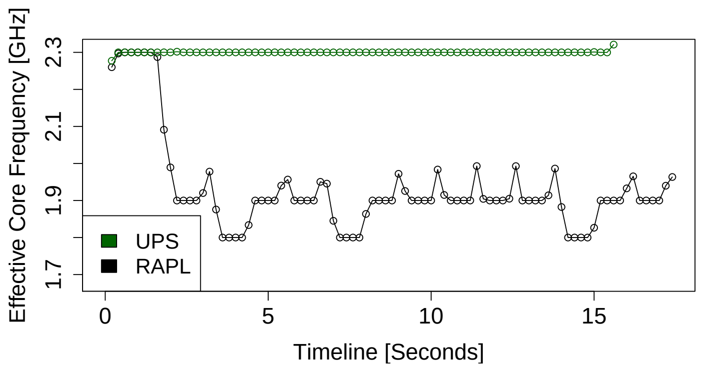


Figure 6.12 Effective core frequency profiles for BT with RAPL and UPS for equal power consumption.

maximum frequency of 2.3GHz.

6.5 Related Work

Chip manufactures have provided various knobs to modulate power directly, e.g., power capping, or indirectly, e.g., dynamic voltage frequency scaling (DVFS), of various components of a server. Conventionally, DVFS has been used to modulate the frequency of cores [Int11]. Intel processors prior to Haswell maintained all cores at a common operating state, i.e., voltage and frequency. Starting with Haswell, Intel introduced per core DVFS. From Sandy Bridge processors onward, Intel introduced running average power limit (RAPL) [Int11], a power capping mechanism implemented in hardware that constrains the average collective power consumption of the cores and the uncore of a processor. A plethora of solutions [Lim06; Rou07; Rou09; Fre05b; Bai15; HF05; Ge07; Gho16; Gho18; Fre05a; Spr06] have been proposed that leverage these knobs to improve the power efficiency of a system. DVFS-based solutions were oblivious of the power consumption of the uncore, which is expected to be a growing component in the future generations of processors [Loh]. RAPL-based solutions off-loaded the management of the uncore's power to Intel's RAPL implementation, which we show is sub-optimal.

Early work exploited DVFS to reduce CPU frequencies during idle time, e.g., due to early arrival at MPI barriers and collectives [Kap05]. An ILP-based approach to model energy [Rou07] was demonstrated to be effective during the runtime of MPI codes to determine optimal power levels for the cores [Rou09]. These works were trying to conserve energy without sacrificing performance by much when utilizing just one core of a node resulting in underutilization of the system. UPS differs in that its foremost objective is to reduce the inefficient power consumption of the uncore of each processor of an HPC job without reducing system utilization, i.e., using all cores of a processor. It conserves power and achieves energy savings by dynamically modulating the frequency of the uncore based on real-time feedback from the system.

Prior work [CM06; Li10] leveraged the effect of concurrency throttling and thread locality to save power and increase performance. An ILP-based runtime approach has been shown to determine how many cores an application should be run on to stay within a given power budget [Tot14; Lan15]. Marathe et al. [Mar15] proposed Conductor, a runtime system that speeds up an application's critical path through an adaptive socket power-allocation algorithm that periodically performs dynamic voltage frequency scaling (DVFS) and dynamic concurrency throttling (DCT) based on application behavior and power usage. Our work differs in terms of granularity and adaptivity. First, we reduce uncore power across resources at processor chip level, exploiting dynamically adaptive feedback methods. Second, UPS does not modify the concurrency or the thread locality, which can be tightly coupled with the application's input.

In recent work, Gholkar et al. [Gho16] presented PTune, a process variation-aware power tuner that uses performance characterization data for all sockets on a cluster and application execution characteristics to minimize the runtime of a job under its power budget. It assumes that the jobs are moldable, i.e., the number of MPI ranks can be varied at the time of job scheduling. UPS does not require jobs to be moldable. It also does not require any prior characterization data to make uncore frequency decisions at runtime. In [Gho18], they proposed PShifter, a feedback-based mechanism that shifts power from processors idling at barriers and collectives to the processors on the critical path to improve performance under a power constraint. PShifter does not strive to save power. Its control calculations depend on the utilization of the cores while being oblivious of the uncore's utilization. Our approach is a dynamic closed-loop feedback controller that specifically monitors the uncore and depending on its utilization reduces any wasteful power consumption by modulating the uncore frequency at runtime. Ellsworth et al. proposed Pow, a system-wide power manager that re-distributed the power budget across a system via scheduling while adhering to a global machine-level power cap [Ell15]. This approach salvaged the unused power but did not address inefficient power usage. Our approach goes one step further by detecting and reducing the wasteful power consumption within a job.

Capacity-improving schemes that increase job throughput have been developed under power limitations by exploiting "hardware overprovisioning", i.e., by deploying more nodes that will be powered at a time [FF05b; Pat13; Sar13a; Sar14; FF05a]. In such a system, the characteristics of a code under strong scaling were used to calculate the optimal number of processors considering core and memory power [Sar13b]. Modifications to the batch scheduler in how small jobs are backfilled depending on their power profile can further increase job capacity [**patki2**]. While these solutions aimed at achieving performance or throughput improvement while utilizing as much of the power budget as possible, our approach aims at conserving power by reducing wasteful power consumption with marginal impact on performance. We do not explicitly aim at *maximizing* the machine's throughput, which is beyond the scope of this paper, but our runtime reduces the overall power and energy footprint of the jobs making more power available for more new jobs to be scheduled on an overprovisioned system.

Hackenberg et al. [Hac15] observed that the uncore frequency depends on the core frequency even when there is no uncore activity. Sundriya et al. [Sun18] compared the impact of uncore frequency scaling to that of core DVFS on power consumption of Gammes, a quantum chemistry application. In other work, a neural network-based uncore frequency scaling approach was simulated for chip multi-core (CMP) power management [Hil]. Unlike this work, we implement and evaluate our runtime system on real hardware, i.e., a Broadwell cluster. Our runtime is suitable for power management of a multi-node and a multi-core system.

Our UPS work is unique in that it is the first of its kind that targets the otherwise neglected uncore component of a chip. It dynamically modulates the frequency of the uncore depending on its utilization measured at runtime. It automatically detects new phases within an application and resets the uncore frequency for each phase. Proposed frameworks like Redfish, the PowerAPI, and Intel's geopm [Red; Gra16; Api] can integrate UPS as a unique closed-loop feedback-based policy for job power management on a cluster. UPS will also relieve the application developers of the burden to explicitly indicate phase changes as required by the APIs (like geopm) as UPS automatically detects phases without any explicit information from the developer.

6.6 Summary

We explored uncore frequency scaling and its impact on performance, power and energy consumption of various HPC applications. To the best of our knowledge, this is the first study of uncore frequency scaling conducted on Broadwell processors. We proposed UPS, a runtime system that automatically modulates the uncore frequency to conserve power without significant performance degradation. As a part of UPS, we also introduced an algorithm that automatically detects phase changes at runtime. Our evaluations indicate that UPS achieves up to 10% energy reduction with less than 1% slowdown. It achieves up to 15% energy savings with a worst case slowdown of 5.5% compared to the default configuration. We also show that UPS achieves up to 20% speedup with proportional energy savings compared to Intel's RAPL with equivalent power usage.

CHAPTER

7 CONCLUSIONS

In this chapter, we summarize the key contributions of the research conducted in this dissertation. We conclude with presenting the future directions of this research.

7.1 Summary

The objective of this work is to maximize the power efficiency of a power-constrained system. We identified power and performance bottlenecks for jobs executing on a power-constrained system. We then proposed strategic solutions that either achieved performance improvements under fixed power budget or achieved power savings with minimal impact of performance.

We started this research with a study of power and performance characteristics of processors, with and without power bounds. We observed a significant variability in the power draw of processors when they achieved uniform peak performance. Under uniform power bounds, this variation in power draw translated to variations in performance. As a result of this variation, an otherwise homogeneous cluster was rendered non-homogeneous under power bounds. We developed Power Partitioner (PPartition) and Power Tuner (PTune), two *variation-aware* power scheduling approaches that achieved improvements in job performance and system performance on such a non-homogeneous power-constrained system. PPartition coordinated with the job scheduler and assigned power budgets to jobs ensuring that the system never exceeded its power budget. For every job, PTune enforced the job-level power budget and maximized its performance by determining the optimal selection of processors and the distribution of the job's power budget across them. PTune and PPartition achieve job speedups of up to 29% and overall system throughput improvements of 5-35% compared to naïve approaches.

We also developed a power-aware cost model to aid in the procurement of a more performant capability system compared to the conventional worst-case power provisioned system. This study reinforced the need for hardware overprovisioning.

We studied the power and performance profiles of typical scientific simulations executing on a supercomputer. The dynamically changing nature of the workload characteristics of these jobs emphasized the need for sophisticated runtime systems to achieve optimal performance for such workloads. Toward this end, we developed Power Shifter (PShifter), a dynamic, feedback-based runtime system that improved job performance under a power constraint by reducing performance imbalance in a parallel job. It used the key concept of a Proportional-Integral-Derivative (PID) controller from control theory to detect the imbalance and shift power from the nodes incurring wait times to the nodes on the critical path, which, in turn, shortened the critical path to improve job performance. PShifter achieved performance improvements of up to 21% and energy savings of up to 23% compared to uniform power allocation. It outperformed static approaches by up to 40% and 22% for codes with and without phase changes, respectively, and outperformed prior dynamic schemes by up to 19%.

We investigated uncore frequency scaling (UFS) as a potential knob for reducing the power footprint of HPC jobs. We presented Uncore Power Scavenger (UPS), a runtime system that leveraged UFS to conserve power during the phases of lower uncore utilization. It performed measurement-based automatic phase detection to dynamically set the best uncore frequency for every phase of execution. UPS achieved up to 10% energy savings with under 1% slowdown. For a worst-case slowdown of 5.5%, it achieved 14% energy savings. UPS achieved up to a 20% speedup and proportional energy savings compared to Intel's RAPL with equivalent power usage, making it a viable solution for power-constrained computing.

Overall, this work presented a conglomerate of power management strategies that improved the power efficiency of the system by either improving performance under a power constraint or by conserving power without any significant impact on performance. Therefore, this work showed that the hypothesis holds.

7.2 Future Work

With the end of Moore's law on the horizon [Sni; Shu; HP], we cannot rely on scaling today's process technology to achieve improvements in power efficiency in the future. Now, more than ever, the research community needs to focus on architectural advances to be able to achieve continued growth in computing. In addition, novel system software solutions need to be employed on power constrained systems to maximize resource utilization and performance under a power budget.

7.2.1 Architectural Designs

In a post Moore era, consecutive generations of processors will possibly belong to the same process technology, so the benefits in performance or power are expected to come from the architecture domain. Toward this end, it is necessary to revisit the conventional node and processor architecture to identify performance bottlenecks for newly emerging classes of applications with the objective of designing architectures that alleviate them.

Conventionally, general purpose CPUs have several identical complex cores, each supporting features such as multi-issue pipelines, branch prediction and out-of-order processing to maximize performance. However, these CPUs are not fully utilized all the time during workload execution. In a post Moore era, transistors will not be free and CMOS real estate will be a design constraint. Hence, we will have to design the sleekest processors by maximizing utilization of transistors on the chip to achieve improvements in performance and power efficiency. Toward this end, we need to evaluate Heterogeneous System Architectures (HSA), such as the “Big Little“ architecture [**biglittle** ; Vil14], that fit the bill. A first step in this direction will be to conduct thorough performance analysis of the state of the art workloads on these architectures. Such architectures under load pose interesting problems such as determining the optimal workload distribution across different types of cores, the choice of cores to execute a workload, and dynamic scheduling of tasks across heterogeneous processing elements to meet the performance and power targets. The ideas presented in this dissertation can be foundational in solving these problems but HSA would provide more and finer knobs to support optimal resource allocation for maximizing the power efficiency of the system.

7.2.2 Resource Contention and Shared Resource Management

Historically, HPC applications were CPU-bound. Over the past years, the performance bottleneck has shifted from CPU to memory, more so with the advent of big data, artificial intelligence, machine learning. The memory hierarchy consists of small private caches, shared resources such as LLC, and memory modules accessed via memory controller and channels. General purpose architecture promotes uniform sharing of resources. However, depending on the characteristics of workloads executing on the cores and their data access patterns, uniform sharing can prove sub-optimal. We need to design “workload-aware“ resource management schemes (such as UPS) that dynamically allocate shared resources such as LLC ways, memory channels or bandwidth to cores to obtain optimal performance. To enable solutions like these, we need to collaborate with chip manufacturers to design and export architectural knobs (e.g., Intel’s Cache Allocation Technology) and measurement infrastructure that support fine-grained resource management.

7.2.3 Runtime Systems

Another important direction of future research is runtime systems for dynamic resource management. We need to explore and evaluate the potential utility of various architectural features in unconventional ways to achieve our objective. As demonstrated in Chapter 5 and Chapter 6, Intel's RAPL and UFS were used as actuation mechanisms in the two proposed runtime systems to achieve the objectives of performance improvement and power conservation, respectively. We need to focus on developing more such light-weight configurable runtime systems for achieving various objectives at different levels of the system hierarchy (such as processor-level, node-level, job-level, cluster-level). Instead of envisioning a single monolithic runtime system managing various resources, the focus should be on employing multiple complimenting runtime systems achieving their respective objectives. An interesting area of investigation would then be to understand how these systems influence and interact with each other.

BIBLIOGRAPHY

[Bai91] Bailey, D. H. et al. “The NAS Parallel Benchmarks”. *The International Journal of Supercomputer Applications* **5**.3 (1991), pp. 63–73.

[Bai15] Bailey, P. E. et al. “Finding the Limits of Power-constrained Application Performance”. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’15. Austin, Texas: ACM, 2015, 79:1–79:12.

[Ber08] Bergman, K. et al. “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems”. Ed. by Kogge, P. 2008.

[Bha17] Bhalachandra, S. et al. “An Adaptive Core-Specific Runtime for Energy Efficiency”. *IPDPS*. 2017, pp. 947–956.

[Bul04] Bulatov, V. et al. “Scalable Line Dynamics in ParaDiS”. *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*. 2004, pp. 19–19.

[Che15] Cheng, H.-Y. et al. “Core vs. Uncore: The Heart of Darkness”. *Proceedings of the 52Nd Annual Design Automation Conference*. DAC ’15. San Francisco, California: ACM, 2015, 121:1–121:6.

[CM06] Curtis-Maury, M. et al. “Online power-performance adaptation of multithreaded programs using hardware event-based prediction”. *Proceedings of the 20th annual international conference on Supercomputing*. ACM. 2006, pp. 157–166.

[Dal11] Dally, B. “Power and programmability: The challenges of exascale computing”. *Presentation at ASCR Exascale Research PI Meeting*. 2011.

[Ecp] “ECP Proxy Apps Suite” (). `http://proxyapps.exascaleproject.org/ecp-suite/`.

[Ell15] Ellsworth, D. A. et al. “POW: System-wide Dynamic Reallocation of Limited Power in HPC”. *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’15. Portland, Oregon, USA: ACM, 2015, pp. 145–148.

[Eti10a] Etinski, M. et al. “Optimizing Job Performance Under a Given Power Constraint in HPC Centers”. *Green Computing Conference*. 2010, pp. 257–267.

[Eti10b] Etinski, M. et al. “Utilization driven power-aware parallel job scheduling”. *Computer Science - R&D* **25**.3-4 (2010), pp. 207–216.

[FF05a] Femal, M. E. & Freeh, V. “Boosting Data Center Performance Through Non-Uniform Power Allocation”. *International Conference on Autonomic Computing*. 2005, pp. 250–261.

[FF05b] Femal, M. E. & Freeh, V. W. “Safe overprovisioning: using power limits to increase aggregate throughput.” *In International Conference on Power-Aware Computer Systems*. 2005.

[Fre05a] Freeh, V. et al. “Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster”. *IPDPS*. 2005.

[Fre05b] Freeh, V. et al. “Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster”. *PPoPP*. 2005, pp. 164–173.

[Ge07] Ge, R. et al. “CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters”. *2007 International Conference on Parallel Processing (ICPP 2007)*. 2007, pp. 18–18.

[Ge10] Ge, R. et al. “PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications” (2010).

[GK06] Gepner, P. & Kowalik, M. F. “Multi-Core Processors: New Way to Achieve High System Performance”. *International Symposium on Parallel Computing in Electrical Engineering (PARELEC’06)*. 2006, pp. 9–13.

[Gho16] Gholkar, N. et al. “Power Tuning HPC Jobs on Power-constrained Systems”. *PACT*. ACM. 2016.

[Gho18] Gholkar, N. et al. “PShifter: Feedback-based Dynamic Power Shifting within HPC Jobs for Performance”. *HPDC*. ACM. 2018.

[Gra16] Grant, R. E. et al. “Standardizing Power Monitoring and Control at Exascale”. *Computer* **49**.10 (2016), pp. 38–46.

[Gup12] Gupta, V. et al. “The forgotten ’uncore’: On the energy-efficiency of heterogeneous cores”. *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. 2012.

[Hac15] Hackenberg, D. et al. “An Energy Efficiency Feature Survey of the Intel Haswell Processor”. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015, pp. 896–904.

[HP] Hennessy, J. L. & Patterson, D. A. “A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development”. `http://iscaconf.org/isca2018/turing_lecture.html`.

[Her09] Heroux, M. A. et al. *Improving Performance via Mini-applications*. Tech. rep. SAND2009-5574. Sandia National Laboratories, 2009.

[Hil] Hill, D. L. et al.

[HF05] Hsu, C.-h. & Feng, W.-c. “A power-aware run-time system for high-performance computing”. *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 2005, p. 1.

[III13] III, J. H. L. et al. “PowerInsight - A commodity power measurement capability”. *IGCC'13*. 2013, pp. 1–6.

[Ina15] Inadomi, Y. et al. “Analyzing and Mitigating the Impact of Manufacturing Variability in Power-constrained Supercomputing”. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015, 78:1–78:12.

[Ins] InsideHPC. “Power Consumption is the Exascale Gorilla in the Room”. `http://insidehpc.com/2010/12/`.

[Int11] Intel. “Intel-64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and 3B: System Programming Guide”. 2011.

[Kap05] Kappiah, N. et al. “Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs.” *SC*. 2005.

[Lan15] Langer, A. et al. “Energy-efficient Computing for HPC Workloads on Heterogeneous Manycore Chips”. *Proceedings of Programming Models and Applications on Multicores and Manycores*. ACM. 2015.

[Li10] Li, D. et al. “Hybrid MPI/OpenMP power-aware computing.” *IPDPS*. Vol. 10. 2010, pp. 1–12.

[Lim06] Lim, M. Y. et al. “Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs”. *SC*. 2006.

[Loh] Loh, G. H. “The cost of uncore in throughput-oriented many-core processors”. *In Proc. of Workshop on Architectures and Languages for Troughput Applications (ALTA)*.

[MW12] Ma, K. & Wang, X. “PGCapping: Exploiting Power Gating for Power Capping and Core Lifetime Balancing in CMPs”. *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 13–22.

[Mar15] Marathe, A. et al. “A Run-Time System for Power-Constrained HPC Applications”. *High Performance Computing*. Ed. by Kunkel, J. M. & Ludwig, T. Cham: Springer International Publishing, 2015, pp. 394–408.

[Mei11] Meisner, D. et al. “Power Management of Online Data-intensive Services”. *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: ACM, 2011, pp. 319–330.

[MG97] Mintchev, S. & Getov, V. "PMPI: High-level message passing in Fortran77 and C". *High-Performance Computing and Networking*. Ed. by Hertzberger, B. & Sloot, P. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 601–614.

[Mud01] Mudge, T. "Power: A first-class architectural design constraint". *Computer* **34**.4 (2001), pp. 52–58.

[Nas] "NASA Advanced Supercomputing Division, NAS Parallel Benchmark Suite v3.3". `http://www.nas.nasa.gov/Resources/Software/npb.html`. 2006.

[Api] "Panel on Power API/Redfish/GEOPM" (). `https://eehpcwg.llnl.gov/documents/conference/sc16/SC16_Laros_Eastep_Benson_API.pdf`.

[Pat13] Patki, T. et al. "Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing". *International Conference on Supercomputing*. Eugene, Oregon, USA, 2013, pp. 173–182.

[Pat15] Patki, T. et al. "Practical Resource Management in Power-Constrained, High Performance Computing". *HPDC*. Portland, Oregon, USA, 2015.

[Pin01] Pinheiro, E. et al. "Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems". *Workshop on compilers and operating systems for low power*. 2001.

[Red] "Redfish API" (). `http:https://www.dmtf.org/standards/redfish/`.

[Rou07] Rountree, B. et al. "Bounding Energy Consumption in Large-Scale MPI Programs". *SC*. 2007, pp. 10–16.

[Rou09] Rountree, B. et al. "Adagio: Making DVS Practical for Complex HPC Applications". *ICS*. 2009.

[Rou12] Rountree, B. et al. "Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound". *IPDPS Workshops*. IEEE Computer Society, 2012, pp. 947–953.

[Sam13] Samih, A. et al. "Energy-efficient interconnect via Router Parking". *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 508–519.

[San11] Sandia National Laboratory. *Mantevo Project Home Page*. `https://software.sandia.gov/mantevo`. 2011.

[Sar09] Sarkar, V. et al. "Software Challenges in Extreme Scale Systems". *Journal of Physics, Conference Series 012045*. 2009.

[Sar13a] Sarood, O. "Optimizing Performance Under Thermal and Power Constraints for HPC Data Centers". PhD thesis. University of Illinois, Urbana-Champaign, 2013.

[Sar13b] Sarood, O. et al. “Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems”. *Proceedings of IEEE Cluster 2013*. Indianapolis, IN, USA, 2013.

[Sar14] Sarood, O. et al. “Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget”. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’14. New Orleans, LA: ACM, 2014.

[Sho14] Shoga, K. et al. “Whitelisting MSRs with msr-safe”. *3rd Workshop on Extreme-Scale Programming Tools at SC*. `http://www.vi-hps.org/upload/program/espt-sc14/vi-hps-ESPT14-Shoga.pdf`. 2014.

[Shu] Shuler, K. “Moore’s Law is Dead: Long Live SoC Designers”. `https://www.design-reuse.com/articles/36150/moore-s-law-is-dead-long-live-soc-designers.html`.

[SC] Smith, C. A. & Corripio, A. “Chapter 6: Control Systems and Their Basic Components”. *Principles and Practice of Automatic Process Control*, pp. 154–192.

[Sni] Snir, M.

[Spr06] Springer, R. et al. “Minimizing execution time in MPI programs on an energy-constrained,power-scalable cluster.” *PPoPP*. 2006.

[SF13] Subramaniam, B. & Feng, W.-c. “Towards Energy-proportional Computing for Enterprise-class Server Workloads”. *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE ’13. Prague, Czech Republic: ACM, 2013, pp. 14–26.

[Sun18] Sundriyal, V. et al. “UNCORE FREQUENCY SCALING VS DYNAMIC VOLTAGE AND FREQUENCY SCALING: A QUANTITATIVE COMPARISON”. *Society for Modeling & Simulation International*. SpringSim-HPC. Baltimore, MD, USA, 2018.

[Top] “Top500 Supercomputer Sites”. `http://www.top500.org/lists/2015/06`. 2015.

[Tot14] Totoni, E. et al. “Scheduling for HPC Systems with Process Variation Heterogeneity”. *Technical Report YCS-2009-443, Department of Computer Science, University of York*. 2014.

[Cap] “Update on the Exascale Computing Project (ECP)”. `http://www.hpcuserforum.com/presentations/santafe2017/MessinaECP.pdf`. 2017.

[Vil14] Villebonnet, V. et al. “Towards Generalizing x0022;Big Little x0022; for Energy Proportional HPC and Cloud Infrastructures”. *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. 2014, pp. 703–710.

[Wat] "WattsUp, power analyzer, watt meter and electricity monitor". `http://moss.csc.ncsu.edu/~mueller/cluster/arc/wattsup/metertools-1.0.0/docs/meters/wattsup/manual.pdf`.

[WM08] Weaver, V. M. & McKee, S. A. "Can hardware performance counters be trusted?" *2008 IEEE International Symposium on Workload Characterization*. 2008, pp. 141–150.

[Yoo03] Yoo, A. et al. "SLURM: Simple Linux Utility for Resource Management". *Job Scheduling Strategies for Parallel Processing*. Vol. 2862. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 44–60.